



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 supervisor

José Antonio Lareo Domínguez

Tutores

Antonio Carlos Domínguez Brito

Enrique Fernández Perdomo

José Daniel Hernández Sosa

15 de mayo de 2013

version 1.0

Agradecimientos

Quisiera dar las gracias a todas aquellas personas que ha contribuido y propiciado que este proyecto haya salido adelante.

En primer lugar a mi familia por estar siempre apoyándome y confiar en mí en todo momento.

A mi tutor Antonio Carlos Brito Domínguez por su inestimable ayuda, sin el cual este proyecto no hubiera sido posible.

A mis amigos, por levantarme el ánimo siempre que lo necesitaba y en especial a mi compañero de proyecto John Wu Wu, con el cual he compartido innumerables horas en las mesas de Informática.

¡Muchas gracias!

Resumen

El presente TFG tiene por objetivo el desarrollo de una librería que permita al usuario controlar de forma sencilla una red de microcontroladores. Como protocolo de comunicación sobre el que trabajar se ha utilizado el bus CAN, que proporciona una capa para el control de errores, configuración del ancho de banda, gestión de prioridades y protocolo de mensajes.

Como resultado al proyecto, se obtiene la librería TouCAN en la cual se establecen dos partes diferenciadas, el lado microcontrolador y el lado supervisor. Cada una de estas partes se desarrollará en un TFG distinto, siendo el lado supervisor el correspondiente a este TFG.

El lado microcontrolador se apoyará sobre la plataforma Arduino. En esta parte, se desarrollará la capacidad de conectar diferentes dispositivos de la red de microcontroladores entre sí, definiendo para ello un protocolo de comunicación que permita la realización de comunicaciones síncronas y asíncronas entre los distintos dispositivos de la red. Para dotar al arduino de la capacidad de hacer uso del protocolo bus CAN, se utilizará un Shield destinado a tal fin.

El objetivo del supervisor será la integración de la red de microcontroladores con dispositivos de propósito general, tales como un ordenador personal, que permita realizar tareas de control y monitorización de los distintos sistemas empotrados situados en la red. Como sistema operativo utilizado en la elaboración de la librería se utilizó una distribución GNU/Linux. Para la comunicación del dispositivo supervisor con la red de microcontroladores se utilizará el puerto serie disponible en la plataforma Arduino.

Palabras clave: Arduino, Microcontrolador, Serial, TouCAN, Supervisor.

Abstract

This TFG aims to develop a library that allow users to easily monitor a network of microcontrollers. As communication protocol it has been used CAN bus, which provides a security layer capable of error control, bandwidth management, prioritization and protocol messages.

As a result of this project, we obtained the TouCan library, which can be divided into two different sides, the microcontroller side and the supervisor side. Each of these parts will be developed in a different TFG, being the supervisor side corresponding to this one.

The microcontroller side will use Arduino as platform. In this part, it will be developed the ability to connect different microcontrollers in the network among each other, defining for it a communication protocol that allows the use of synchronous and asynchronous communication. To give the Arduino the capacity to use the CAN bus protocol, we will use a shield.

The aim of the supervisor is the integration of the microcontroller network with general purpose devices, such as a personal computer, that allows monitoring and control tasks of the various embedded systems located in the network. The operating system used in the development of the library was a GNU / Linux distribution. For the communication between the supervisor and the microcontroler network, we will use the serial port available in the Arduino platform.

Keywords: Arduino, Microcontroller, Serial, TouCAN, Supervisor.

Índice General

1. Introducción	1
1.1. Estado Actual	1
1.2. Objetivos	3
2. Análisis del Problema	5
2.1. ¿Qué es el bus CAN?	5
2.1.1. Capa física	6
2.1.1.1. Baja velocidad tolerante a fallos	7
2.1.1.2. Alta velocidad	7
2.1.1.3. Reloj, sincronización, bit timing	8
2.1.2. Capa de enlace de datos	9
2.1.2.1. Tipos de trama	9
2.1.2.2. Trama de datos y trama remota	10
2.1.2.3. Gestión de Acceso al Bus	11
2.1.2.4. Filtrado de mensajes, enmascaramiento	12
2.2. Arduino	13
2.3. El MCP2515	15
2.3.1. El módulo CAN	15
2.3.2. La lógica de control	16
2.3.3. El bloque de protocolo SPI	16
2.3.3.1. Juego de Instrucciones	17
2.3.4. Banco de Registros	21
2.3.4.1. Registros asociados a la transmisión de tramas	21
2.3.4.1.1. TXBnCTRL - Registro de control para el buffer de transmisión n	21
2.3.4.1.2. TXBnSIDH - Buffer de transmisión n. Parte alta del identificador estándar	22
2.3.4.1.3. TXBnSIDL - Buffer de transmisión n. Parte baja del identificador estándar	22
2.3.4.1.4. TXBnEID8 - Buffer de transmisión n. Parte alta del identificador extendido	22
2.3.4.1.5. TXBnEID0 - Buffer de transmisión n. Parte baja del identificador extendido	23

ÍNDICE GENERAL

2.3.4.1.6.	TXBnDLC - Buffer de transmisión n. Longitud del campo de datos	23
2.3.4.1.7.	TXBnDm. Buffer de transmisión n. Byte m	23
2.3.4.2.	Registros asociados a la recepción de tramas	24
2.3.4.2.1.	RXB0CTRL - Control del buffer de recepción 0	24
2.3.4.2.2.	RXB1CTRL - Control del buffer de recepción 1	25
2.3.4.2.3.	RXBnSIDH - Buffer de recepción n. Parte alta del identificador estándar	25
2.3.4.2.4.	RXBnSIDL - Buffer de recepción n. Parte baja del identificador estándar	26
2.3.4.2.5.	RXBnEID8 - Buffer de recepción. Parte alta del identificador extendido	26
2.3.4.2.6.	RXBnEID0 - Buffer de recepción. Parte baja del identificador extendido	27
2.3.4.2.7.	RXBnDLC - Buffer de recepción. Longitud del campo de datos	27
2.3.4.2.8.	RXBnDM - Buffer de recepción n. Byte m	27
2.3.4.3.	Registros asociados a filtros	28
2.3.4.3.1.	RXFnSIDH - Filtro n. Parte alta del identificador estándar	28
2.3.4.3.2.	RXFnSIDL - Filtro n. Parte baja del identificador estándar	28
2.3.4.3.3.	RXFnEID8 - Filtro n. Parte alta del identificador extendido	29
2.3.4.3.4.	RXFnEID0 - Filtro n. Parte baja del identificador extendido	29
2.3.4.4.	Registros asociados a máscaras	30
2.3.4.4.1.	RXMnSIDH - Máscara n. Parte alta del identificador estándar	30
2.3.4.4.2.	RXMnSIDL - Máscara n. Parte baja del identificador estándar	30
2.3.4.4.3.	RXMnEID8 - Máscara n. Parte alta del identificador extendido	31
2.3.4.4.4.	RXMnEID0 - Máscara n. Parte baja del identificador extendido	31
2.4.	TouCAN	31
2.4.1.	Topología	31
2.4.2.	Comunicaciones	32
2.4.3.	Tramas	32
2.4.4.	Estructuras	33
2.4.4.1.	node	33
2.4.4.2.	tCAN	35
2.4.5.	Métodos	35
2.4.5.1.	Métodos Internos	35
2.4.5.2.	Métodos del Usuario	37

ÍNDICE GENERAL

3. Competencias	39
3.1. CII01	39
3.2. CII02	39
3.3. CII04	39
3.4. CII18	40
4. Aportaciones	41
5. Normativa y Legislación	43
5.1. Normativa	43
5.1.1. Ley de Protección de Datos	43
5.1.2. Código tipo	43
5.1.3. Inscripción de un programa informático	44
5.2. Licencias	44
5.2.1. GNU GPL	44
5.2.2. LGPL	44
5.2.3. Creative Commons	45
6. Pliego de Condiciones	47
6.1. Objeto de este pliego	47
6.2. Pliego de Condiciones Generales	47
6.3. Pliegos de especificaciones técnicas	47
6.3.1. Especificaciones de materiales, equipos y software	47
6.3.2. Especificaciones de ejecución	47
6.4. Pliego de Clausulas Administrativas Particulares	48
6.5. Licencia de Uso	48
7. Metodología y Plan de Trabajo	49
8. Requisitos	53
8.1. Hardware	53
8.2. Software	53
9. Diseño e Implementación	55
9.1. Diseño e Implementación	55
9.1.1. Comunicación lado del microcontrolador	56
9.1.2. Comunicación lado del PC	57
9.1.2.1. Estructuras	58
9.1.2.1.1. communication_struct	58
9.1.2.1.2. nodo_info	59
9.1.2.1.3. node_list	59
9.1.2.2. Funciones Privadas	59
9.1.2.3. Funciones Públicas	60
9.1.3. Protocolo de comunicación	63
9.1.4. Mecanismo de reintentos	64

ÍNDICE GENERAL

10. Pruebas	65
10.1. Pruebas Funcionales	65
10.1.1. Petición Síncrona a un nodo común.	65
10.1.2. Petición Asíncrona a un nodo común.	66
10.1.3. Petición Síncrona al nodo maestro.	68
10.1.4. Petición Asíncrona a un nodo maestro.	68
10.1.5. Petición Síncrona y Asíncrona a un nodo común.	69
10.1.6. Petición Síncrona y Asíncrona a un nodo maestro.	70
10.1.7. Petición Síncrona y Asíncrona a un nodo común y a un nodo maestro.	70
10.2. Pruebas de Rendimiento	71
11. Conclusiones	73
11.1. Trabajo Futuro	74
12. Manual de Usuario	75
12.1. Instalación	75
12.1.1. Lado Controlador	75
12.1.2. Lado Supervisor	77
12.2. Compilación y Ejecución	78
12.2.1. Lado Controlador	78
12.2.2. Lado Supervisor	80

Capítulo 1

Introducción

1.1. Estado Actual

Las redes de interconexión han sido a lo largo de la evolución de la informática, un tema recurrente de estudio. Por una parte se busca que el sistema sea lo más rápido y fiable posible y por otra que se minimizen al máximo los costes derivados del mismo. En este TFG, se mostrará el trabajo realizado entorno al protocolo de comunicación bus CAN[1], (Controlled Area Network) mediante la elaboración de un supervisor que permita controlar en un PC, una red de microcontroladores.

Desarrollado por Robert Bosch GmbH, se trata de un protocolo de comunicación[2] basado en una topología bus[3], es decir, los nodos se interconectan en serie a través de un único bus. Presentado en 1986 y orientado inicialmente al sector automovilístico, supuso un gran impacto en dicha industria, en la cual ya se presentaban problemas entre los diferentes componentes disponibles en un automóvil debido a que los coches se volvían cada vez más complejos, y con un mayor número de dispositivos provocando que las comunicaciones se tornaran cada vez más complejas y costosas.

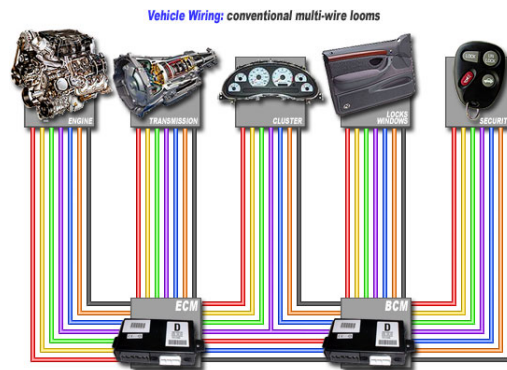


Figura 1.1: Conexión de los componentes de un automóvil antes del bus CAN. Fuente: <http://www.canbuskit.com/>.

El protocolo bus CAN no sólo solucionaba los problemas de implementación, comunicación y coste entre los diferentes dispositivos sino que también añadía una capa de seguridad capaz de realizar control de errores, ancho de banda, gestión de prioridades y protocolo de mensajes. Por todas estas razones, el protocolo bus CAN se estableció con éxito en el sector automovilístico.

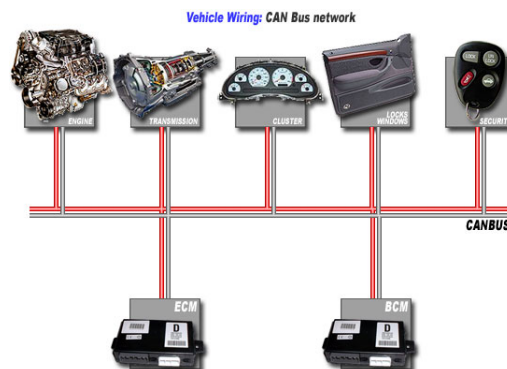


Figura 1.2: Conexión de los componentes de un automóvil al usar bus CAN. Fuente: <http://www.canbuskit.com/>.

Del protocolo bus CAN podemos encontrar dos especificaciones del estándar[4], la 2.0A (estándar) y la 2.0B (extendido), de los cuales no sólo se benefician la industria automovilística, sino también el de la medicina, aeronáutica, etc.

Hoy en día, los microcontroladores[5], se encuentran más extendidos de lo que pueda parecer, elementos tan cotidianos como una nevera o elementos más complejos como un robot, se encuentran contruidos a base de diferentes microcontroladores destinados a realizar diferentes tareas. En el caso de los robots por ejemplo, se plantea la necesidad de coordinar diferentes sensores capaces de dotarlo de cierta autonomía, para lo cual deberán estar comunicados entre ellos. Por otra parte, podría presentarse también la necesidad de monitorizar determinado

sensor y actuar en consecuencia.

1.2. Objetivos

Debido a que el uso de microcontroladores está en auge y las características que provee el protocolo bus CAN, surge la idea crear la API TouCAN con el fin de dotar a los usuarios, una API amigable, fácil de usar y robusta.



Figura 1.3: Propuesta de logo para TouCAN.

TouCan se compone principalmente de dos partes:

- **Lado microcontrolador:** consistente en la comunicación entre los diferentes microcontroladores.
- **Lado supervisor:** consistente en la comunicación el PC y la red de microcontroladores.

El trabajo de este proyecto fin de grado estará enfocado en el lado del supervisor, sin embargo, para llevar a cabo la implementación será necesario el previo estudio de la librería TouCAN que funcionará en el lado del microcontrolador así de como su modificación y adaptación del código para el lado supervisor.

Como objetivos académicos, se decidieron aplicar en la medida de lo posible los conocimientos obtenidos a lo largo de la titulación, de entre los cuales se destacan los siguientes:

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

A continuación se muestra como se ha estructurado el documento y cada una de las partes que lo compone:

- **Análisis del problema:** en este capítulo se profundizará en el conocimiento del protocolo bus CAN así como de la librería TouCAN.
- **Competencias:** contendrá cada una de las competencias que se deberán abarcar en el proyecto y se especifican los apartados en los cuales se ven satisfechas.
- **Aportaciones:** se llevará a cabo un análisis del impacto que supone el proyecto a nuestro entorno socio-económico.
- **Normativa y legislación:** se expondrán las distintas normativas y legislaciones que afectan al proyecto.
- **Pliego de condiciones:** en este apartado se analiza el pliego de condiciones establecido tanto a nivel particular como a nivel general.
- **Metodología / Plan de trabajo:** aquí se expondrá detalladamente la metodología que se ha seguido a la hora de desarrollar el TFG y así mismo la planificación que se ha llevado a cabo para el cumplimiento de los objetivos.
- **Requisitos:** se especifican los requerimientos tanto a nivel de hardware como a nivel de software asociados al proyecto.
- **Diseño e Implementación:** todos los detalles relativos a las modificaciones llevadas a cabo a la API TouCAN así mismo como del supervisor en el PC estarán en este capítulo.
- **Pruebas:** se presentan las pruebas realizadas con el fin de demostrar la correctitud del código.
- **Conclusiones:** por último se relatará en este capítulo las experiencias y conocimientos adquiridos a lo largo del desarrollo del proyecto.
- **Manual de Usuario:** apartado dedicado a orientar al usuario en la instalación y utilización del software.

Capítulo 2

Análisis del Problema

Como ya se adelantó en la introducción, el objetivo de este TFG está orientado al trabajo sobre el protocolo bus CAN, y más en concreto en la API TouCAN. Para ello, antes deberemos obtener unos conocimientos básicos del estándar CAN que nos permitan abordar posteriormente la API TouCAN. Por otra parte, se hará una introducción teórica del hardware que usaremos en la etapa de diseño y por último se analizará la librería TouCAN.

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[4] 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

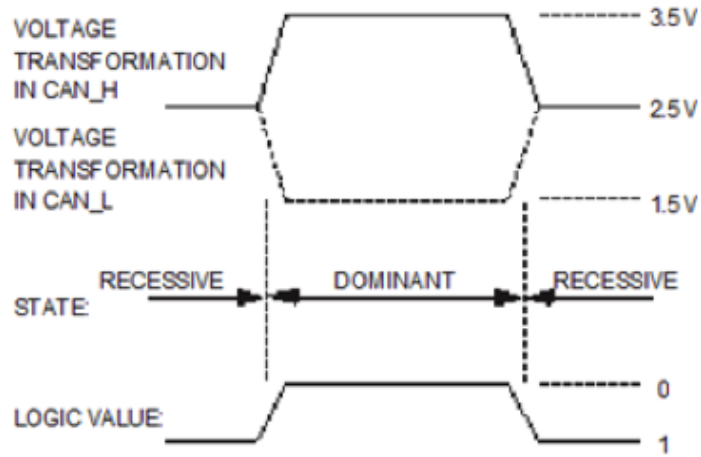


Figura 2.1: Niveles eléctricos para la definición del valor dominante y recesivo (baja velocidad)

Es necesario que cada uno de los transceptores[6] 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

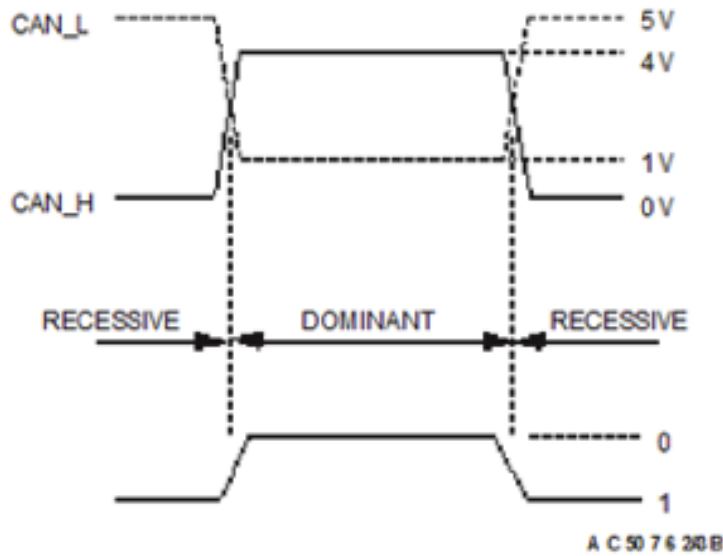


Figura 2.2: Niveles eléctricos para la definición del valor dominante y recesivo (alta velocidad)

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:

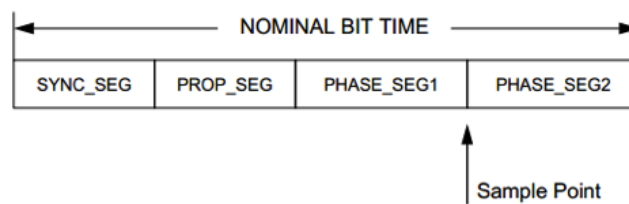


Figura 2.3: Fases de muestreo de bit.

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.

<i>Etapas</i>	<i>Longitud en cuanto</i>	<i>Descripción</i>
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:

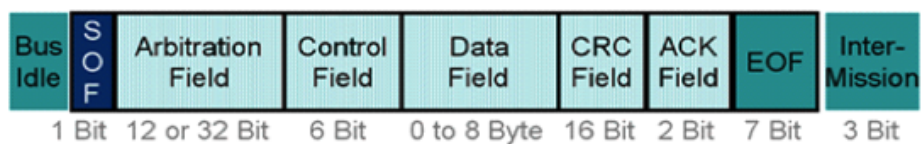


Figura 2.5: Bits de trama de datos y trama remota. Fuente: <http://www.softing.com/>

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 Length 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:

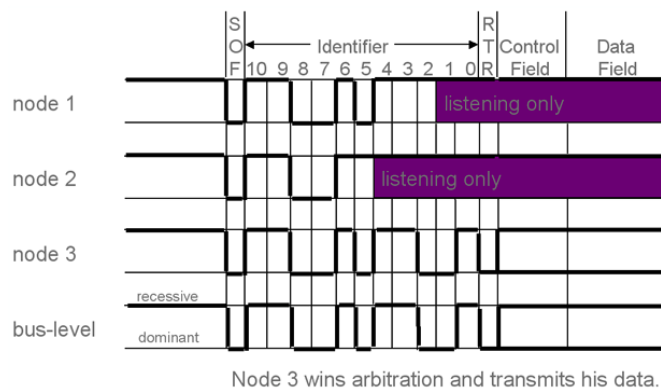


Figura 2.7: Ejemplo de arbitraje de bus: El nodo 3 gana el arbitraje. Fuente: <http://www.softing.com/>

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[7] por su versatilidad, amplio soporte de la comunidad y la relación cantidad precio. En este sentido nos decantamos por la placa **Arduino UNO**[8] ya que era la que más se ajustaba a nuestros intereses.



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

Una de las virtudes de Arduino es que sus características son ampliables mediante otras placas denominadas **shields**[9], 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**[10], el cual se basaba en un **PIC MCP2515**[11] como controlador para el protocolo CAN. Así pues, para poder desarrollar sobre esta plataforma fue necesario conocer su configuración interna.

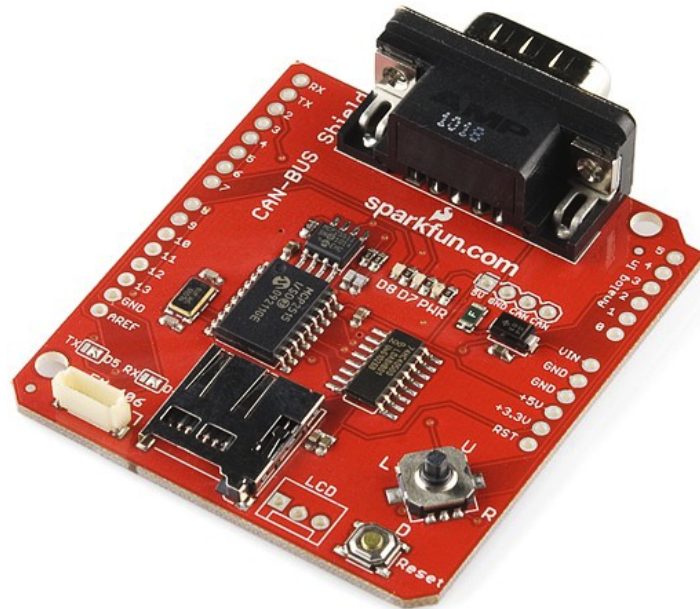


Figura 2.11: Shield "Sparkfun CAN Shield". Fuente: <http://reflexiona.biz/>

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

2.3. EI 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.

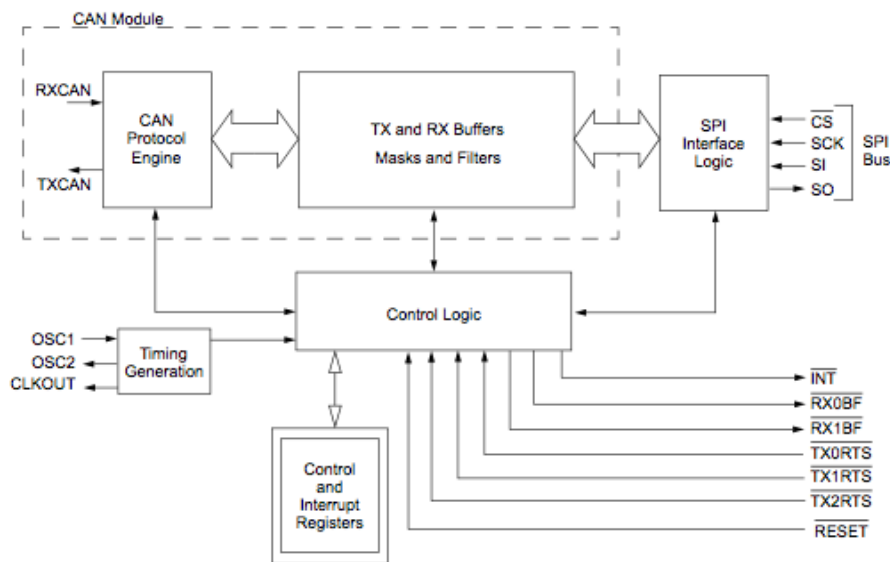


Figura 2.12: Diagrama de los principales bloques del MCP2515. Fuente: *Microchip MCP2515 Datasheet*.

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 este 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) también son válidos para comprobar la correcta recepción de un mensaje.

De forma adicional, existen también tres pines disponibles para iniciar inmediatamente 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[12] 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.

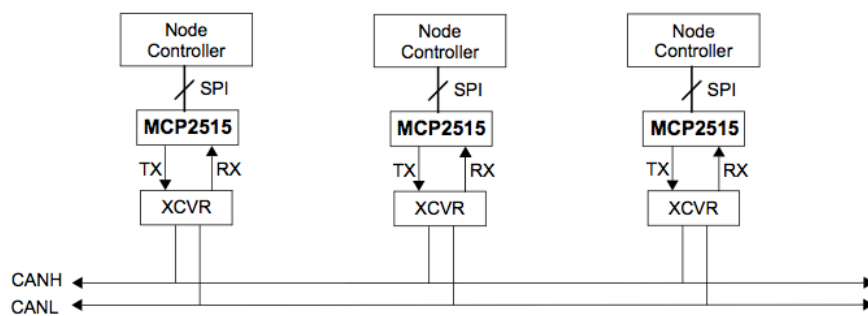


Figura 2.13: Ejemplo de red CAN: El microcontrolador conectado al MCP2515 y éste, a su vez, conectado al bus. Fuente: Microchip MCP2515 Datasheet.

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 0nmn	Se usa en el comienzo de una transmisión de un mensaje.
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 cronogramas correspondientes a las instrucciones SPI. Sin embargo, para su comprensión, 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.

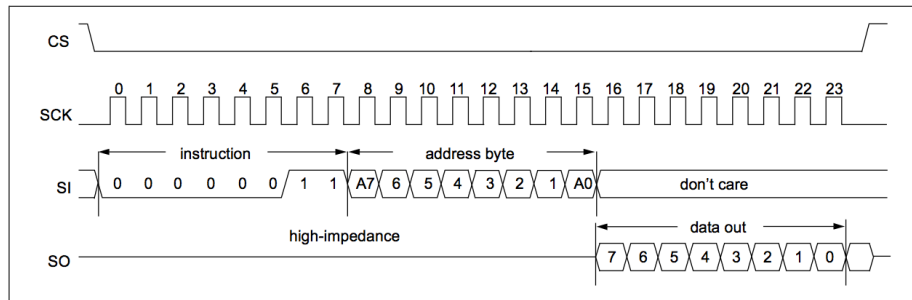


Figura 2.14: Cronograma correspondiente a la operación READ de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

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.

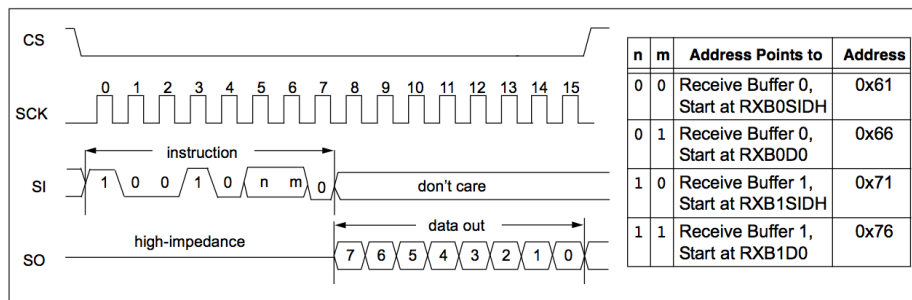


Figura 2.15: Cronograma correspondiente a la operación READ RX BUFFER de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

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.

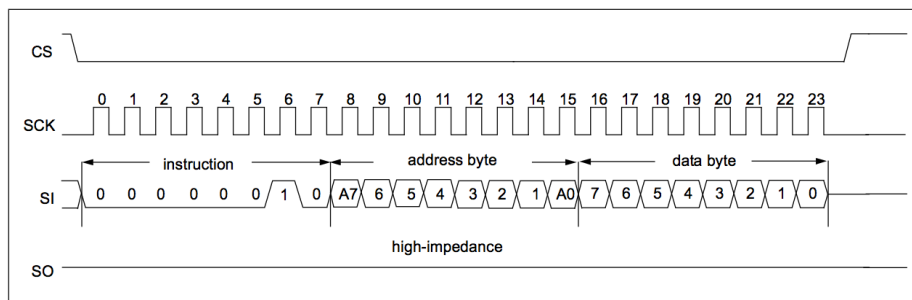


Figura 2.16: Cronograma correspondiente a la operación BYTE WRITE Buffer de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

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.

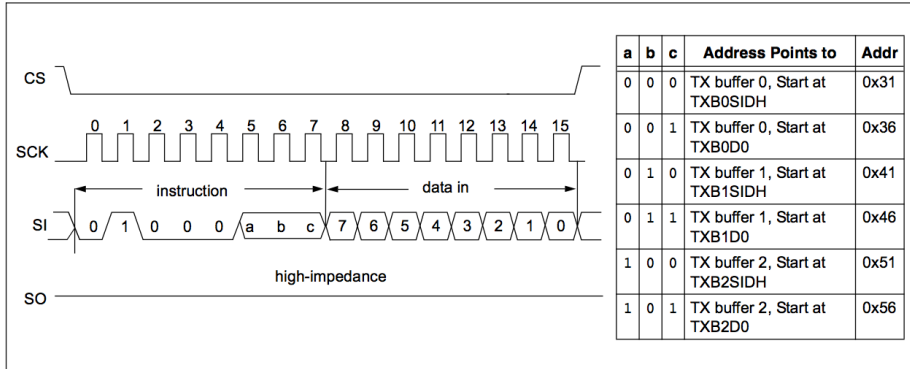


Figura 2.17: Cronograma correspondiente a la operación LOAD TX BUFFER de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

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.

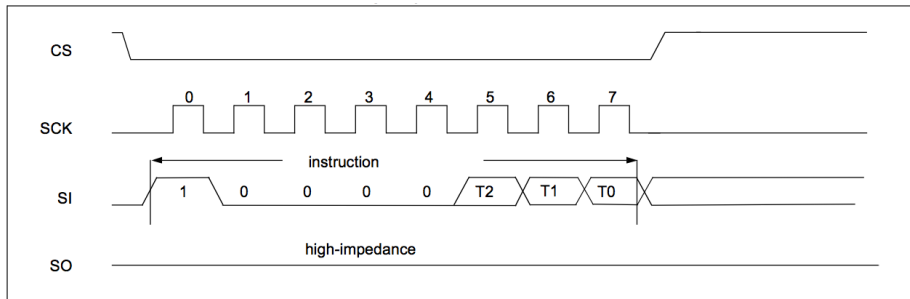


Figura 2.18: Cronograma correspondiente a la operación REQUEST-TO-SEND (RTS) de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

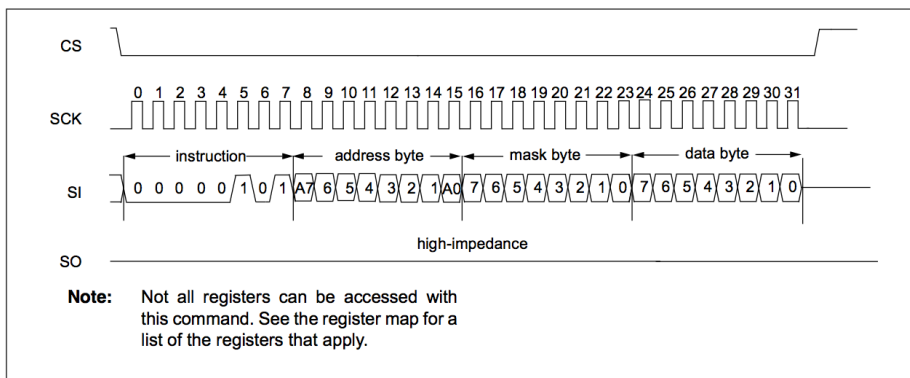


Figura 2.19: Cronograma correspondiente a la operación BIT MODIFY de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

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.

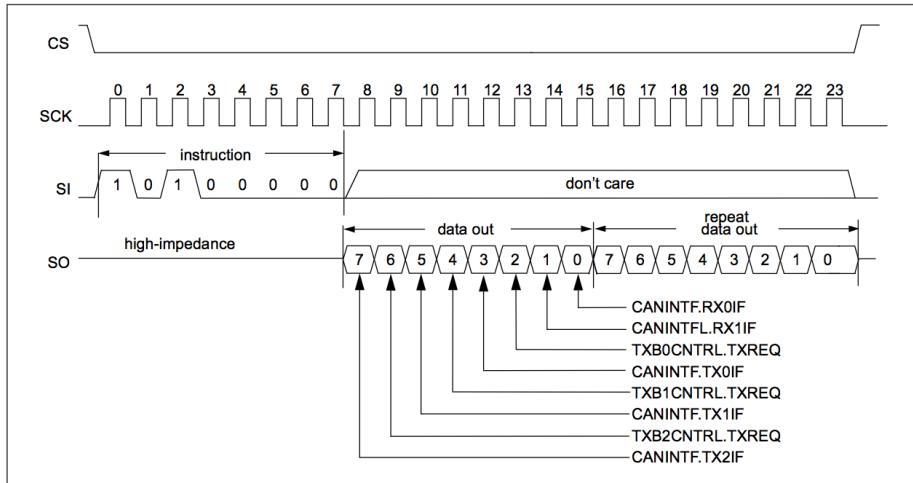


Figura 2.20: Cronograma correspondiente a la operación READ STATUS de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

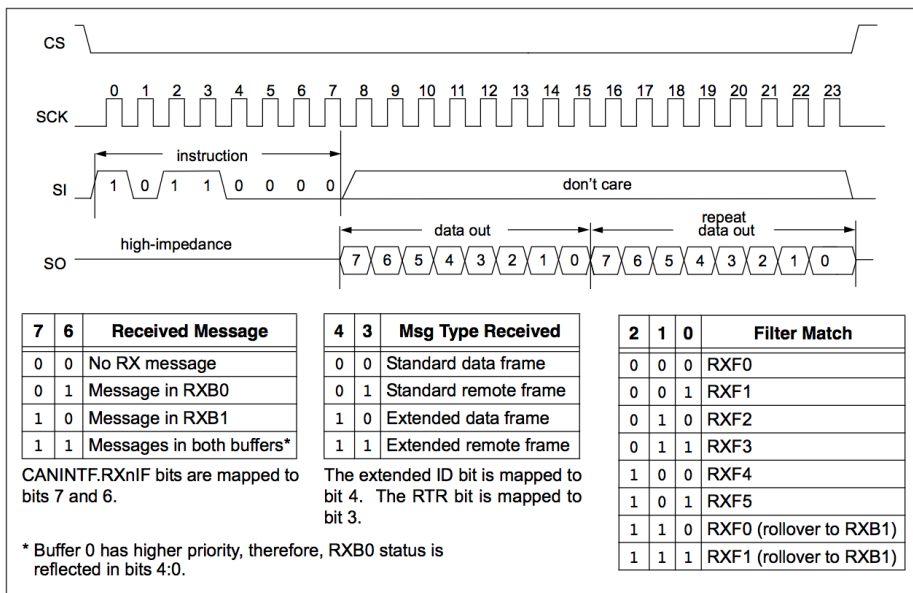


Figura 2.21: Cronograma correspondiente a la operación RX STATUS de la Interfaz SPI. Fuente: Microchip MCP2515 Datasheet.

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 transmisión 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 que 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 exitosa.
- 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 desea realizar un envío de trama.
 - 1 = El buffer se encuentra actualmente pendiente de una transmisión (El microcontrolador 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 microcontrolador 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 parte 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 parte 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 registros 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 estándares 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	FILHITO
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 estándares 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” explicaremos 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” explicaremos 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 estándar.
- 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” explicaremos 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” explicaremos 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 : 16 > 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.

2.4. TouCAN

TouCAN se trata de una API para microcontroladores basada originalmente en la librería Arcan[13] a la cual se le añadió una capa superior dotándola de un sencillo protocolo de comunicación. Ésta API nace de la necesidad de disponer de una librería que no supusiera un esfuerzo al usuario usarla. Dada la carga de trabajo que llevaría desarrollarla en un solo TFG, se decide dividir el trabajo de la siguiente manera: Lado microcontrolador la cual se recoge en el TFG “Integración de redes de microcontroladores distribuidos basados en bus CAN. Lado microcontrolador” por John Wu Wu y el lado supervisor que se abarca en este TFG. Debido a que ambos proyectos están íntimamente ligados, se hace necesario tener una noción básica de la librería para luego comprender el funcionamiento del lado supervisor. A continuación se describen las estructuras y los métodos de los que dispone TouCAN.

2.4.1. Topología

TouCAN define tres tipos de elementos que pueden coexistir en el protocolo. En primer lugar nos encontramos con los nodos comunes, son dispositivos que están destinados a proporcionar datos a otros dispositivos, es por ello que no tienen iniciativa propia y tan sólo cumplen el rol de esclavo. Por otra parte nos

encontramos a los nodos maestros, los cuales tendrán la capacidad de pedir datos a otros nodos, y además responder ante peticiones de otros nodos maestros.

En la siguiente figura, se puede apreciar una estructura de ejemplo:

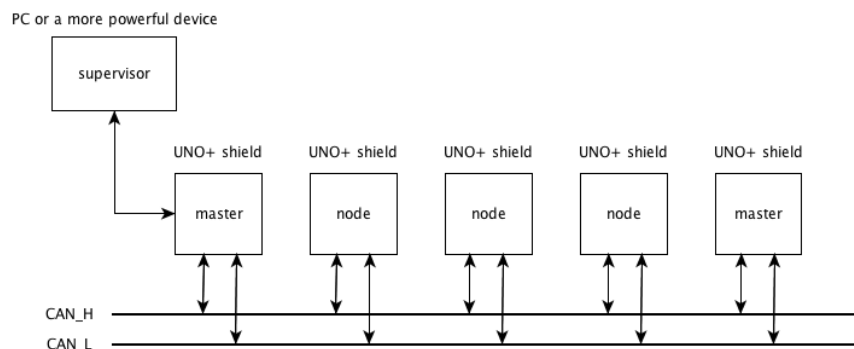


Figura 2.45: Esquema de la red TouCAN

Como vemos, un supervisor debe ir conectado a un nodo maestro ya que es el único que puede realizar peticiones a otros nodos.

2.4.2. Comunicaciones

TouCAN define dos tipos de comunicaciones posibles:

- **Síncrona:** En este tipo de comunicación se garantiza la recepción de la respuesta de una determinada petición. Su uso está destinado a situaciones en las que se quiera recibir un dato puntual de un dispositivo dado.
- **Asíncrona:** Una comunicación asíncrona proporciona un flujo constante de información al dispositivo que ha realizado la petición (hasta que se cancele la petición). Al contrario que la petición síncrona, en este tipo de comunicación no se garantiza la recepción de cada uno de los mensajes. Este tipo de comunicación resulta imprescindible cuando se desea recibir información de forma constante de un determinado dispositivo.

2.4.3. Tramas

Los distintos tipos de tramas se pueden clasificar según el tipo de comunicación.

Comunicaciones síncronas:

- **Request (petición):** Su función es la de pedir un dato a un dispositivo determinado. Solamente puede ser usada por nodos maestros.
- **Answer (respuesta):** Se trata de una trama de respuesta a una trama de tipo Request. Puede ser usada tanto por nodos maestros como por nodos comunes.

Comunicaciones asíncronas:

- **Start (comienzo):** Trama usada para pedir a un nodo el envío de forma periódica de datos. El periodo deseado se deberá especificar en el primer byte del mensaje. Solamente puede ser usada por nodos maestros.
- **Trama acknowledge (reconocimiento):** Se trata de una trama de reconocimiento a una petición asíncrona. Puede ser usada por nodos maestros y nodos comunes.
- **Trama de bulk (envío masivo):** Se trata de la trama enviada una vez se ha realizado un Start y se ha contestado con un Acknowledge a dicha petición. Por tanto esta trama contendrá el dato requerido en la petición. Estas tramas serán enviadas de forma periódica al nodo maestro que realizó la petición. Tanto los nodos maestros como los nodos esclavos puede hacer uso de la trama.
- **End (finalización):** Con esta trama se finaliza la operación de envío de datos de forma asíncrona. Esta trama solo puede ser utilizada por un nodo maestro.

A continuación se describe un tipo de trama presente en los dos tipos de comunicaciones:

- **Reject:** Este tipo de trama permitirá rechazar una determinada petición, informando al dispositivo de la razón. Tanto nodos maestros como nodos comunes pueden utilizarla.

2.4.4. Estructuras

2.4.4.1. node

- **Código:**

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

- **Descripción:** Esta estructura almacenará para un nodo de la red, información asociada a comunicaciones síncronas y asíncronas. Con esta estructura, un

nodo maestro podrá llevar un control de cada nodo y de las comunicaciones llevadas a cabo. En el maestro, se deberá declarar un vector de nodos, donde cada posición se corresponderá con un nodo de la red. La última posición del vector se destina al propio nodo maestro. Para ahorrar tiempo en la búsqueda de elementos en la tabla se establecieron dos tipos de identificadores: reales y virtuales.

- **Identificadores reales:** Son los conocidos de antemano por el usuario administrador de la red. Representan el valor real de identificador que posee cada dispositivo.
 - **Identificadores virtuales:** Se asignan en función de la posición del nodo en la tabla del maestro.
- **Campos:** A continuación se describen cada uno de los campos por lo que está constituida la estructura. Hay que destacar que para la última posición del vector de nodos, es decir la que se corresponde con la del propio dispositivo maestro, los significados de algunos campos varían.
- **S_supervisor_op:** Indica si la operación síncrona ha sido realizada por el maestro tomando el valor de false, o si la ha realizado el supervisor por medio del maestro siendo el valor true.
 - **S_timestamp:** Marca de tiempo de una determinada petición síncrona. Indicará si para un determinado nodo se ha realizado alguna petición síncrona.
 - **S_answer:** En este campo se llevará el control de recepción de una petición de tipo Request. Si S_answer es 0 indicará que no se ha recibido ninguna respuesta, si es 1 indicará que se ha recibido un Answer y si es un 2 entonces se habrá recibido un Reject del Request. Cuando se trata de la posición correspondiente al nodo Maestro, el significado de Answer se usará para saber si se ha recibido una petición de otro maestro. Siendo 0 el caso en el que no ha producido una petición y 1 cuando si ha producido.
 - **S_length:** Indica el tamaño del vector de datos correspondientes a una operación síncrona.
 - **S_data[8]:** Contiene los datos correspondientes recibidos de una operación síncrona.
 - **A_superviso_op:** Indica si la operación asíncrona ha sido realizada por el maestro siendo false en tal caso, o si la ha realizado el supervisor por medio del maestro tomando el valor de true.
 - **A_timestamp:** Marca de tiempo de una determinada petición asíncrona. Indicará si para un determinado nodo se ha realizado alguna petición asíncrona.
 - **A_ack:** Este campo llevará el control de la peticiones asíncronas. Si su valor es 0 significará que no hemos recibido respuesta, si es 1 habremos recibido una confirmación del nodo al que se ha realizado la petición y si es 2 significará que se ha rechazado la operación con una trama de tipo Reject. Cuando se trata de la posición correspondiente al Maestro, 0 indicará que no se ha producido una petición asíncrona y 1 indicará una petición por parte de otro maestro.

- **written:** A la hora de realizar una petición asíncrona. Si el periodo establecido es muy pequeño puede darse la situación de que se sobreescriban datos debido a que al dispositivo no le ha dado tiempo a procesarlo. Con este campo se indicará si se sobreescribió algún valor.
- **A_length:** Indica el tamaño del vector de datos correspondientes a una operación asíncrona.
- **A_data[8]:** Contiene los datos correspondientes recibidos de una operación asíncrona.

2.4.4.2. tCAN

- **Código:**

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

- **Descripción:** Esta estructura que representa un mensaje CAN.
- **Campos:**
 - **id_s:** Se corresponde con el identificador fuente del mensaje.
 - **id_d:** Identificador de destino del mensaje.
 - **frametype:** Tipo de trama.
 - **header:** Se trata de una estructura compuesta por los campos rtr y length. Rtr indicará si se trata de una trama remota (1) o de datos (0), mientras que en length se almacenará el tamaño del vector de datos.
 - **data[8]:** Vector de datos.

2.4.5. Métodos

A continuación se listan las funciones privadas y públicas de las que dispone TouCAN.

2.4.5.1. Métodos Internos

- **tTouCAN(void):** constructor de la clase TouCAN.
- **boolean check_message(void):** Comprueba la existencia de mensajes en los buffers 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 buffers 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.
- **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 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 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 do_start_transmission(tCAN *, int, int, int, float, int)**: 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 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 is_reject(tCAN *)**: Pregunta si el mensaje es un reject, devolviendo 1 en caso afirmativo y 0 en caso contrario.

- **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.

2.4.5.2. Métodos del Usuario

- **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.
- **int init_nodes(node *, int):** Inicializa la lista de nodos del maestro.
- **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.
- **int request(tCAN *, int, int, int, int, node *, int, int):** Es la función de usuario para realizar una petición síncrona.
- **int answer(tCAN *, int, int, int, node *, int, int):** Es la función de usuario para realizar una respuesta síncrona.
- **int start_transmission(tCAN *, int, int, int, int, float, float, node *,int):** Es la función de usuario para realizar un comienzo de comunicación asíncrona.
- **int acknowledge(tCAN *, int, int, int):** 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 stop_transmission(tCAN *, int, int, int, node *, int):** 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.

- **void get_start(tCAN *, int, void (*time_handler)(), int (*sync_callback)(int, byte *, int), byte *)**: 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 *msje, int source, int period, int (*sync_callback)(int, byte *, int), int (*async_callback)(int, byte *, int))**: 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 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 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 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 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().

Capítulo 3

Competencias

A continuación se describen como han sido cubiertas cada una de las competencias asociadas al proyecto 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, Análisis, Diseño e Implementación a través de los cuales se van mostrando cada una de las decisiones tomadas y el proceso que se ha llevado a cabo hasta el desarrollo final.

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 de metodología y planificación se muestra la organización llevada a cabo para la obtención de los diferentes metas marcadas. Por otra parte, dentro del capítulo Aportaciones se lleva a cabo la valoración del impacto del proyecto a nuestro entorno socio-económico.

3.3. CII04

Capacidad para elaborar el pliego de condiciones técnicas de una instalación informática que cumpla los estándares y normativas vigentes.

Esta competencia se abarca dentro del capítulo Pliego de Condiciones en la cual se especifican cada una de las condiciones asociadas al proyecto.

3.4. 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

En la actualidad, el sector de la robótica está en constante progreso. A medida que la tecnología avanza, los prototipos de robots se vuelven cada vez más complejos con un mayor número de sensores. Sin lugar a duda, la complejidad en el diseño se ve aumentada, y se hace imprescindible buscar el diseño más óptimo con el fin de reducir al máximo el presupuesto. Este problema no se presenta de forma exclusiva únicamente en el sector robótico, sin ir mas lejos, dentro del desarrollo de la domótica también nos podemos encontrar en la misma tesitura.

Este proyecto puede suponer una gran ayuda y ahorro en el ámbito de las comunicaciones entre microcontroladores por medio de la red CAN, propiciando la aparición de nuevos proyectos que se beneficien de la simplicidad y comodidad de las librerías desarrolladas. Por otra parte, dado que se trabaja sobre plataformas Arduino, este proyecto puede impulsar el uso de ellas con las ventajas que ello aporta (asequibles, multiplataformas y de código abierto).

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 General 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 software 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

Se trata de una licencia parecida 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 eficaz 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.*



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 supervisor de redes de microcontroladores distribuidos touCAN.

6.2. Pliego de Condiciones Generales

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

- Capacidad de supervisar un red de microcontroladores bajo la librería touCAN.
- 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 y posibilidad para representarlos.

6.3. Pliegos de especificaciones técnicas

6.3.1. Especificaciones de materiales, equipos y software

Las especificaciones de todos los materiales de los cuales se compone el proyecto se describe ampliamente en el apartado de requisitos, en la cuál 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 16980 euros.

Recurso	Coste
Programador (120 horas)	40 euros/hora.
Analista Funcional (60 horas)	50 euros/hora.
Jefe de Proyecto (150 horas)	60 euros/hora.
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

Como metodología de desarrollo se utilizó las técnicas de desarrollo software propias del Proceso Unificado (Unified Process). 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 se compone de cuatro 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.

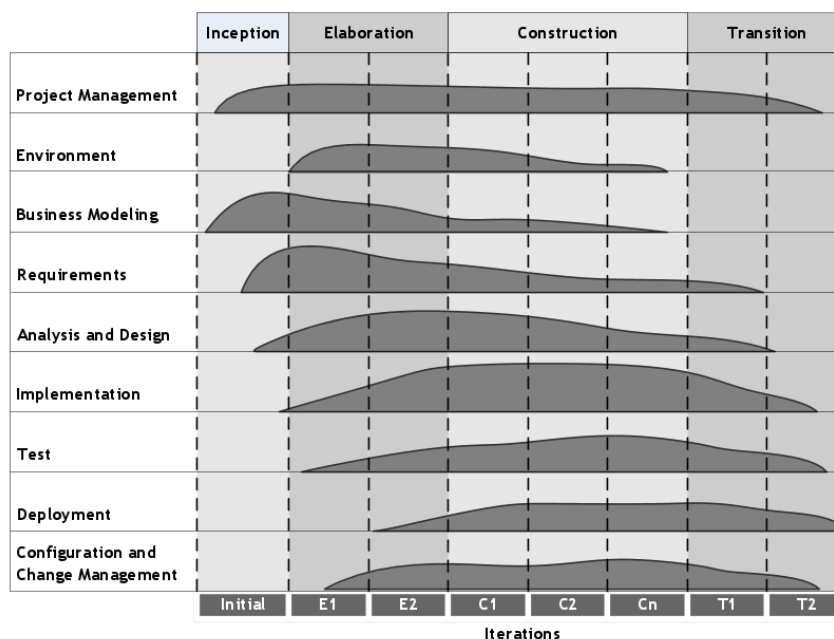


Figura 7.1: Evolución de las diferentes etapas de un proyecto desarrollado mediante el Proceso Unificado. Fuente: <http://es.wikipedia.org/>

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

1. **Familiarización Arduino y Bus CAN.** Estudio, familiarización y análisis de la arquitectura de desarrollo para sistemas empujados basada en microcontroladores Arduino. Estudio, familiarización y análisis del bus de comunicaciones CAN y su integración con la mencionada arquitectura de sistemas empujados.
2. **Diseño y desarrollo en el lado supervisor.** 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 supervisor.** 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.** Confeción del Documento de Trabajo de Fin de Grado a partir de toda la documentación realizada en las diferentes etapas en las que se ha organizado. Preparación de la Defensa.

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 supervisor	100 horas
Etapa 3: Diseño y desarrollo de prototipo para el lado 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 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.**
6. **PC.** En concreto se ha utilizado un PC con CPU Intel Core 2 Duo a 2Ghz, Memoria ram de 2GB y una tarjeta gráfica GeForce 9400M.

8.2. Software

A nivel de software fue necesario:

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

Capítulo 9

Diseño e Implementación

9.1. Diseño e Implementación

En la implementación cabe distinguir dos partes claramente diferenciadas, la parte de implementación que involucra a los microcontroladores, y la parte que involucra al PC. Para la comunicación entre el PC y el nodo supervisor se hará uso del protocolo serie, debido a que todas las placas Arduino tienen integrado al menos un puerto serie[14]. En el caso específico de los microcontroladores usados, Arduino Uno rev3, poseen sólo un puerto serie cuyos pines correspondientes son el 0 (RX) y el 1 (TX). Estos pines se encuentran soldados al puerto usb del microcontrolador. A continuación se puede apreciar el diagrama correspondiente al microcontrolador.

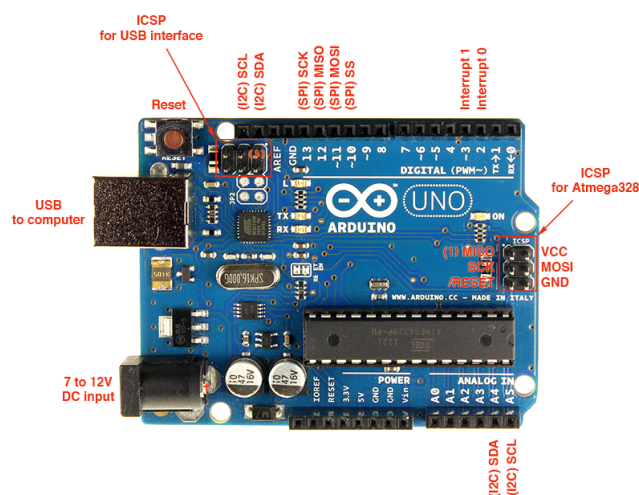


Figura 9.1: Arduino Uno rev3. Fuente: <http://cefire.edu.gva.es/>

Así pues para comunicarnos con el microcontrolador, a nivel de hardware tan

sólo deberemos conectarnos haciendo uso del cable usb.

9.1.1. Comunicación lado del microcontrolador

Como ya se comentó, para la comunicación con el PC se hará uso del puerto serie disponible. A nivel de software, Arduino nos provee de librerías[15] con funciones de lectura y escritura en éste puerto, con lo cual nos resuelve la problemática de tener que implementarlas. A continuación se listan y describen cada una de las funciones disponibles:

- **begin(speed):** Con esta función se establecerá la velocidad con la cual los bits serán transmitidos. Resulta imprescindible que en la comunicación entre el microcontrolador y el PC, ambos trabajen a la misma velocidad sobre el puerto serie.
- **end():** Desactiva el puerto serie, permitiendo que los pines correspondientes (0,1) puedan volver a ser usados como pines digitales para otros fines.
- **available():** Indica que han llegado datos en el puerto serie y están disponibles en el buffer, listos para ser leídos. Hay que destacar que el buffer tiene un tamaño de 128 bytes.
- **read():** Permite la lectura de un byte del buffer de datos del puerto serie.
- **flush():** Se descartan aquellos valores del buffer de entrada que aún no han sido leídos.
- **print(val), print(val,format):** Envía al puerto serie los datos pasados por parámetros como caracteres ASCII.
- **println(val), println(val,format):** Función similar a print, pero que además añade un salto de línea final a la ristra de caracteres enviados.
- **write(val), write(str), write(buf,len):** Permite el envío de datos al puerto serie. A efectos prácticos tiene la misma funcionalidad que print.

Una vez asimilado el funcionamiento de la comunicación serial en el microcontrolador se procedió a la modificación de la librería TouCAN con el fin de dotarla de rutinas que faciliten la comunicación con el PC. A continuación se analizan cada una de las funciones/estructuras que fueron modificadas y/o añadidas:

- **node struct:** En esta estructura, fue necesario añadir los campos S_supervisor_op (peticiones síncronas) y A_supervisor_op (peticiones asíncronas). Como ya se explicó en el apartado de análisis, en esta estructura se almacenan los datos de las peticiones. Dado que ahora puede darse el caso de que la petición sea del supervisor, puede no interesarnos guardar el mensaje sino por el contrario reenviarle la respuesta a PC por el puerto serie. Es por ello que se necesitaba reflejar de alguna manera en el caso en el que nos encontramos.
- **request:** En esta función se añadió el parámetro is_supervisor_op con el fin de distinguir cuando se trata de un envío por parte del propio nodo maestro o una petición por parte del supervisor en cuyo caso se reflejaría poniendo en

la posición correspondiente de la estructura nodes, el campo S_supervisor_op a true. Así pues, una vez llegada la respuesta, tan solo deberemos comprobar dicho campo, y actuar en consecuencia.

- **start_transmission:** Al igual que con el request, fue necesario añadir el campo is_supervisor_op. Si se trata de una petición del supervisor, se deberá poner el parámetro A_supervisor_op a true.
- **stop_transmission:** Las modificaciones llevadas a cabo son iguales que las llevadas a cabo en el start_transmission.
- **save_data:** En esta función, a la hora de procesar el mensaje recibido, se tuvo que añadir la comprobación de si la petición la ha realizado el nodo supervisor o si la ha realizado el propio maestro. Para ello se comprobará el campo A_supervisor_op/S_supervisor_op(según sea el caso). En el caso de que se trate de una petición del supervisor, evitamos guardar el mensaje en la estructura, y se opta por enviarle la respuesta al PC por el puerto serie.
- **check_received_command:** Esta función se encarga de comprobar si hay algún comando enviado por el PC en el puerto serie. Para ello, se llama a la función Serial.available() del puerto Serie. En caso de que encuentre datos, los irá leyendo byte a byte hasta completar el comando.
- **send_frame:** Esta operación se encarga de enviar al PC la trama de datos correspondiente a la petición que haya hecho.
- **perform_supervisor_operation:** Esta función es la encargada de comprobar si hay algún comando del PC (haciendo uso de la función check_received_command), y si así fuera, se encargaría de procesarlo. A la hora de procesar un comando, se ha de considerar si se trata de una petición para un nodo de la red, o si por el contrario se trata de una petición para el propio nodo supervisor. De tal forma que si se tratara para un nodo, se hace uso de las funciones propias de envío de tramas a la red de microcontroladores. Con lo cual una vez enviada la trama y recibida la respuesta del nodo correspondiente, se reenvía dicha trama por el puerto serie al PC. Si por el contrario se trata de una petición para el propio nodo supervisor, simplemente se deberá obtener el dato requerido por el PC y enviárselo.
- **send_device_list:** Con esta función, se envía al PC la tabla de dispositivos de la red de microcontroladores, con el fin de que éste sepa la lista de nodos disponibles a los cuales puede realizar peticiones. Hay que destacar que dado que se pueden realizar peticiones al propio nodo maestro al que está conectado el supervisor, también se envía su id. Se decidió establecer la última posición de la tabla como la correspondiente al nodo maestro.

9.1.2. Comunicación lado del PC

En esta parte, se partió de la librería serial desarrollada por Tod E. Kurt[16], la cual inicialmente se encontraba implementada en C (Para GNU/Linux), y cuya programación estaba orientada a ser usada en forma de comando. La comunicación con el microcontrolador, a pesar de que pueda parecer complicada, es más sencilla de lo que parece. Cuando se conecta el microcontrolador al PC, automáticamente

se crea un fichero que se corresponde con el dispositivo. La comunicación desde este lado básicamente consistirá en la apertura, lectura y escritura en dicho fichero. Obviamente hay que tener en cuenta una serie de parámetros de configuración a la hora de la apertura los cuales ya se encuentran contemplados en la librería. Se escogió ésta librería frente a otras disponibles principalmente por su simplicidad y la no necesidad de tener que instalarla. Se comenzó creando una clase en C++ de dicha librería para posteriormente modificar y añadirle funciones específicas necesarias en la comunicación con los microcontroladores. A continuación se describe la clase TouCAN correspondiente al nodo supervisor.

9.1.2.1. Estructuras

9.1.2.1.1. `communication_struct`

- **Descripción:** Se trata del buffer interno en las comunicaciones con el microcontrolador.

- **Campos:**

- **S_timestamp:** Marca de tiempo en petición síncrona.
- **S_length:** Tamaño vector de datos de la petición síncrona.
- **S_data:** Vector de datos síncrono.
- **S_destination:** Identificador de destino de la petición síncrona.
- **S_source:** Identificador fuente de la petición síncrona.
- **S_type:** Tipo de la petición síncrona.
- **A_timestamp** Marca de tiempo de la petición asíncrona.
- **A_length_input** Tamaño del vector de entrada de la petición asíncrona.
- **A_length_output** Tamaño del vector de salida de la petición asíncrona.
- **A_data_input:** Vector de datos de entrada de la petición asíncrona.
- **A_data_output:** Vector de datos de salida de la petición asíncrona.
- **A_destination:** Identificador de destino de la petición asíncrona.
- **A_source:** Identificador fuente de la petición asíncrona.
- **A_type:** Tipo de la petición asíncrona.

- **Código:**

```
typedef struct {
    struct timeval S_timestamp; //synchronous part
    int S_length;
    unsigned char S_data[MAX_SIZE];
    int S_destination;
    int S_source;
    int S_type;

    struct timeval A_timestamp; //asynchronous part
    int A_length_input;
    int A_length_output;
    unsigned char A_data_input[MAX_SIZE];
```

```

    unsigned char A_data_output [MAX_SIZE];
    int A_destination;
    int A_source;
    int A_type;

}comunicacion_struct;

```

9.1.2.1.2. nodo_info

- **Descripción:** Contiene información sobre un nodo de la red de microcontroladores. En concreto, almacena su identificador, y el número de reintentos que lleva en conexiones síncronas y asíncronas.
- **Campos:**
 - **id:** Identificador del dispositivo.
 - **S_ntries:** Número de reintentos en peticiones síncronas.
 - **A_ntries:** Número de reintentos en peticiones asíncronas.

- **Código:**

```

typedef struct {
    int id;
    int S_ntries;
    int A_ntries;
}node_info;

```

9.1.2.1.3. nodo_list

- **Descripción:** Estructura que contiene un vector de estructuras node_info.
- **Campos:**
 - **devices:** Lista de dispositivos.
 - **length:** Tamaño de la lista de dispositivos.

- **Código:**

```

typedef struct {
    node_info devices [MAX_NUM_DEVICES];
    int length;
}node_list;

```

9.1.2.2. Funciones Privadas

- **int serialport_writebyte(uint8_t b):** Escribe el byte pasado por parámetro en el puerto serie.
Parámetros:
 - **b:** Byte a escribir en el puerto serie.**Retorna:** '0' éxito, '-1' fallo.

- **int serialport_write(const char* str):** Escribe una ristra en bytes en el puerto serie.

Parámetros:

- **str:** Ristra a escribir en el puerto serie.

Retorna: '0' éxito, '-1' fallo.

- **int serialport_readbyte(int timeout):** Lee del puerto serie un byte.

Parámetros:

- **timeout:** Tiempo máximo de espera.

Retorna: '0' éxito, '-1' timeout.

- **int serialport_read_until(char until):** Lee del puerto serie hasta encontrar el carácter pasado por parámetro until.

Parámetros:

- **until:** Carácter de espera.

Retorna: '0' éxito, '-1' fallo.

- **char *serialport_value():** Retorna el puntero al buffer donde se almacena los datos leídos del puerto serie.

Parámetros:

- **ninguno.**

Retorna: Dirección del buffer.

9.1.2.3. Funciones Públicas

- **int serialport_init(const char* serialport, int baud):** Inicializa el dispositivo.

Parámetros:

- **serialport:** Ruta del fichero correspondiente con el dispositivo serie.
- **baud:** Baudios.

Retorna: descriptor del fichero en caso de éxito, '-1' caso de fallo.

- **int serialport_reinit(const char* serialport, int baud):** Función que reinicializa un dispositivo serie. Con el parámetro serialport se especifica el fichero correspondiente al dispositivo y en baud se especifican los baudios.

Parámetros:

- **serialport:** Ruta del fichero correspondiente con el dispositivo serie.
- **baud:** Baudios.

Retorna: '0' éxito, '-1' fallo.

- **serialport_close():** Función que cierra la conexión con el dispositivo serie al que está conectado.

Parámetros:

- **ninguno.**

Retorna: '0' éxito, '-1' fallo.

- **bool check_device(int timeout):** Esta función comprueba si el dispositivo serie está listo, con la cual nos servirá para realizar el "handshake" ó apretón de manos entre el nodo maestro y el supervisor.

Parámetros:

- **timeout:** Tiempo máximo de espera.

Retorna: 'READY' éxito, 'NOT READY' fallo.

- **bool get_device_list(node_list *list):** Obtiene del maestro la lista de nodos disponibles.

Parámetros:

- **list:** Lista de nodos.

Retorna: 'true' éxito, 'false' fallo.

- **int print_device_list(node_list *list):** Método que imprime la lista de nodos disponibles.

Parámetros:

- **list:** Lista de nodos.

Retorna: '0' lista con elementos, '-1' lista sin elementos.

- **int send_command(int type, int destination, int period, int length, unsigned char data[], communication_struct *communications, node_list *list):** Función que se encarga de enviar un comando al puerto serie.

Parámetros:

- **type:** Tipo de comando (STRT, STOP ó REQU).
- **destination:** Identificador del dispositivo de destino.
- **period:** Periodo en caso de tratarse de una petición de tipo STRT.
- **length:** Longitud del vector de datos.
- **data:** Vector de datos.
- **communications:** Buffer interno.
- **list:** Lista de nodos.

Retorna: 'COMMAND_SENT' comando enviado, 'PENDING_TASK' comando no enviado debido a una tarea pendiente, 'DEVICE_ERROR' el dispositivo de destino presenta errores, 'INVALID_COMMAND' el comando es inválido.

- **int check_received_frame(int timeout, communication_struct *communications, node_list *list):** Función que comprueba si se ha recibido una trama del microcontrolador.

Parámetros:

- **timeout:** Tiempo máximo de espera.
- **communications:** Buffer interno.
- **list:** Lista de nodos.

Retorna: 'ANSW' trama answer recibida, 'REJE' trama reject recibida, 'BFRM' bulk frame recibido, 'ACKN' acknowledge recibido, 'NOT_RECEIVED_FRAME' no se ha recibido ninguna trama, 'UNKNOWN_FRAME' se ha recibido una trama desconocida, 'ERROR_TIMEOUT' se ha producido time out en medio de una recepción, 'ERROR_FRAME' error en trama.

- **int received_frame(int timeout, communication_struct *communications, node_list *list):** Función que guarda en el buffer interno la trama recibida.

Parámetros:

- **timeout:** Tiempo máximo de espera.
- **communications:** Buffer interno.
- **list:** Lista de nodos.

Retorna: 'ANSW' trama answer recibida, 'REJE' trama reject recibida, 'BFRM' bulk frame recibido, 'ACKN' acknowledge recibido, 'NOT_RECEIVED_FRAME' no se ha recibido ninguna trama, 'UNKNOWN_FRAME' se ha recibido una trama desconocida, 'ERROR_TIMEOUT' se ha producido time out en medio de una recepción, 'ERROR_FRAME' error en trama.

- **void print_frame(int checkvalue, communication_struct *communications):** Función que imprime el buffer interno.

Parámetros:

- **checkvalue:** Hay que pasarle el valor que retorna la función check_frame_type.
- **communications:** Buffer interno.

Retorna: nada.

- **void init_comunication_struct(communication_struct *communications):** Función que inicializa el struct de comunicaciones.

Parámetros:

- **communications:** Buffer interno.

Retorna: nada.

- **int check_timestamp(int timeout, communication_struct *communications, node_list *list):** Comprueba si se ha superado el tiempo máximo de espera en alguna solicitud. De ser así, se reenvía de nuevo el comando. En caso de que se superen los reenvíos máximos se notifica en la tabla de dispositivos.

Parámetros:

- **timeout:** Tiempo máximo de espera.
- **communications:** Buffer interno.
- **list:** Lista de nodos.

Retorna: 'SYNCH_TIMEOUT' timeout en petición síncrona, 'ASYNCH_TIMEOUT' timeout en petición asíncrona, 'SYNCH_ERROR' error en el dispositivo de la petición síncrona, 'ASYNCH_ERROR' error en el dispositivo de la petición asíncrona.

9.1.3. Protocolo de comunicación

En el envío de comandos/tramas del supervisor al microcontrolador y viceversa, se tuvo que especificar y añadir una capa superior de comunicación al protocolo serie, de tal forma que ambas partes se pudieran comunicar. En este tipo de comunicaciones, toma gran relevancia que ambos dispositivos se sincronicen en algún punto de la ejecución, también denominado "handshake" ó apretón de manos. Una vez que los dispositivos se han sincronizado, significará que el microcontrolador estará a la escucha de posibles peticiones.

Debido a que el supervisor antes de poder enviar comandos, necesita disponer de la lista de dispositivo presentes en la red de microcontroladores, se estableció que una vez realizado el apretón de manos, el microcontrolador maestro conectado al supervisor, se la enviara de la siguiente forma:

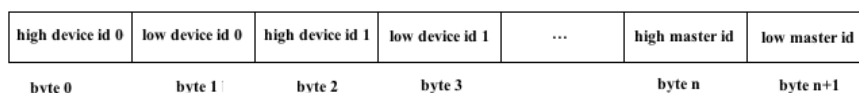


Figura 9.2: Transmisión de la tabla de dispositivo.

Asimismo, en el lado del PC se elaboró un función que se encargara de la lectura de la tabla, la cual debe ser llamada un vez producida la sincronización. Un detalle en la implementación que se tuvo que tener en cuenta fue la limitación a nivel de byte en las escrituras y lecturas (limitación en las funciones que proporciona el microcontrolador). En el caso de la id se presentaba un problema, y es que el sistema está diseñado para que se soporten muchos más de 256 dispositivos (en concreto hasta 2048). Para resolver este problema simplemente se tuvo que tener

en cuenta que a la hora de leer/escribir la id, se hará partiendo la id en dos bytes. De tal forma que se estableció que a la hora de escribir, primero se enviará el byte más significativo, para luego enviar el byte menos significativo.

Por otra parte, respecto al protocolo de los comandos, dado que todos los comandos poseen un formato uniforme, tan sólo se tuvo que definir un orden de envío de cada uno de los bits que componen un comando. El cual se corresponde con el siguiente:

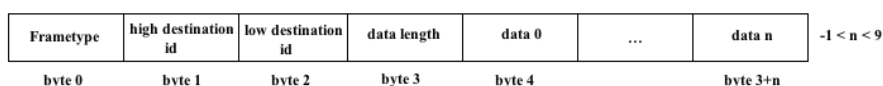


Figura 9.3: Transmisión de un comando.

Como vemos, lo primero que se envía es el tipo de trama al que se corresponde la petición, seguidamente se envía el id del dispositivo al que está dirigida la petición, para luego enviar el tamaño del vector de datos y por último cada uno de los datos del vector. Tanto en el envío como en la recepción de datos se seguirá dicho orden de lectura/escritura, de tal forma que se irá construyendo el mensaje según se vayan leyendo los bytes.

9.1.4. Mecanismo de reintentos

Como forma de asegurar que las peticiones son realizadas, se decide implementar un mecanismo de reintentos de peticiones, de tal forma que si un usuario realiza una petición, y salta el timeout establecido, internamente se volverá a reenviar la trama. El número de reintentos máximos queda definido en la macro "MAX.RETRIES", la cual por defecto tiene el valor de 4. En caso de que se superen el número máximo de reintentos, el dispositivo se marcará como inválido, de forma que el usuario sepa que existe algún tipo de problema con el nodo que quiere conectar. Así pues, por medio de este sistema aportamos una mayor robustez a la librería frente a posibles pérdidas de peticiones y por otra parte, conseguimos identificar posibles nodos en la red que presenten alguna anomalía.

Capítulo 10

Pruebas

En este apartado se ha decidido dividir las pruebas en dos grandes grupos: “Pruebas Funcionales” y “Pruebas de rendimiento”. A continuación se analizará con mayor detenimiento cada grupo.

10.1. Pruebas Funcionales

El objetivo de las pruebas funcionales es el de testar cada una de las funcionalidades y demostrar su correcto funcionamiento. Para ello se describirán diversos casos y se analizarán los resultados obtenidos. Para las pruebas se ha decidido utilizar una cuenta incremental en el primer campo de datos con el fin de comprobar que no se pierda ningún paquete.

10.1.1. Petición Síncrona a un nodo común.

En esta prueba se pretende comprobar la correctitud a la hora de realizar peticiones síncronas desde el supervisor a un nodo de la red de controladores.

Como primera prueba, se harán una serie de peticiones continuas al nodo común 1 obteniendo el siguiente resultado:

```

osl1@osl1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
osl1@osl1:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Answer frame, Source = 1 Size 8: 0 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 1 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 2 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 3 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 4 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 5 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 6 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 7 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 8 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 9 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 10 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 11 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 12 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 13 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 14 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 15 1 2 3 4 5 6 7

```

Figura 10.1: Peticiones síncronas a nodo común.

Se puede observar la recepción consecutivamente de las respuestas del nodo sin ninguna pérdida.

10.1.2. Petición Asíncrona a un nodo común.

En esta prueba se pretende comprobar la correctitud a la hora de realizar peticiones asíncronas desde el supervisor a un nodo de la red de controladores.

Para la realización de la prueba, se realiza un start al nodo común 1 (usando como periodo 150ms).

```

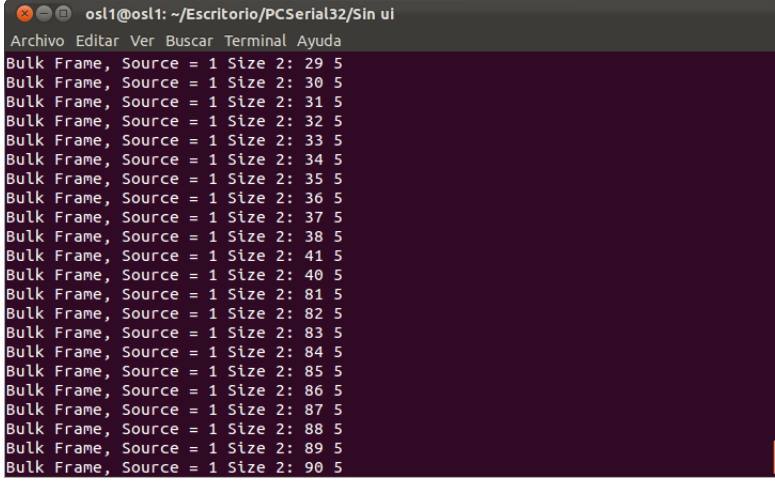
osl1@osl1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
osl1@osl1:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 1
Bulk Frame, Source = 1 Size 2: 0 5
Bulk Frame, Source = 1 Size 2: 1 5
Bulk Frame, Source = 1 Size 2: 2 5
Bulk Frame, Source = 1 Size 2: 3 5
Bulk Frame, Source = 1 Size 2: 4 5
Bulk Frame, Source = 1 Size 2: 5 5
Bulk Frame, Source = 1 Size 2: 6 5
Bulk Frame, Source = 1 Size 2: 7 5
Bulk Frame, Source = 1 Size 2: 8 5
Bulk Frame, Source = 1 Size 2: 9 5
Bulk Frame, Source = 1 Size 2: 10 5
Bulk Frame, Source = 1 Size 2: 11 5
Bulk Frame, Source = 1 Size 2: 12 5
Bulk Frame, Source = 1 Size 2: 13 5
Bulk Frame, Source = 1 Size 2: 14 5

```

Figura 10.2: Petición asíncrona a un nodo común.

Al realizar la petición, se recibe una trama del tipo ack aceptándonos pues la petición asíncrona y recibiendo los mensajes de tipo bulk correspondientes.

A continuación se muestra el comportamiento al realizar un start al nodo común 1 (usando como periodo 150ms) y desconectar la dispositivo maestro durante la recepción de mensajes de tipo bulk.



```

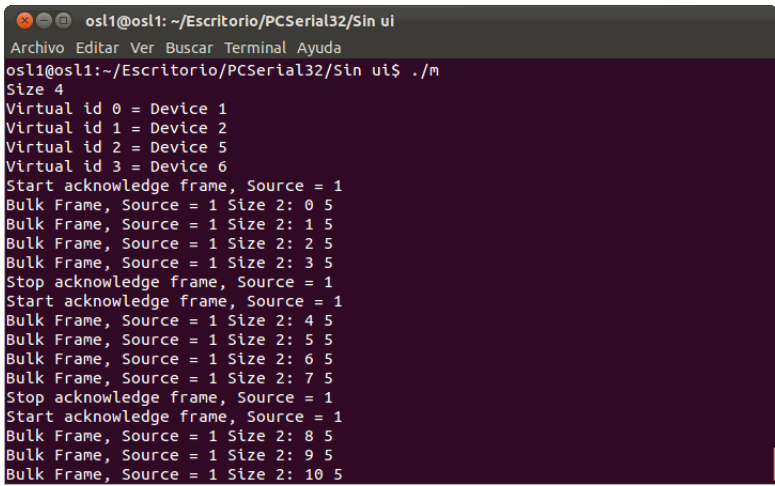
osl1@osl1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
Bulk Frame, Source = 1 Size 2: 29 5
Bulk Frame, Source = 1 Size 2: 30 5
Bulk Frame, Source = 1 Size 2: 31 5
Bulk Frame, Source = 1 Size 2: 32 5
Bulk Frame, Source = 1 Size 2: 33 5
Bulk Frame, Source = 1 Size 2: 34 5
Bulk Frame, Source = 1 Size 2: 35 5
Bulk Frame, Source = 1 Size 2: 36 5
Bulk Frame, Source = 1 Size 2: 37 5
Bulk Frame, Source = 1 Size 2: 38 5
Bulk Frame, Source = 1 Size 2: 41 5
Bulk Frame, Source = 1 Size 2: 40 5
Bulk Frame, Source = 1 Size 2: 81 5
Bulk Frame, Source = 1 Size 2: 82 5
Bulk Frame, Source = 1 Size 2: 83 5
Bulk Frame, Source = 1 Size 2: 84 5
Bulk Frame, Source = 1 Size 2: 85 5
Bulk Frame, Source = 1 Size 2: 86 5
Bulk Frame, Source = 1 Size 2: 87 5
Bulk Frame, Source = 1 Size 2: 88 5
Bulk Frame, Source = 1 Size 2: 89 5
Bulk Frame, Source = 1 Size 2: 90 5

```

Figura 10.3: Petición asíncrona a un nodo común.

Como se puede observar, llega un punto en el que se deja de recibir la secuencia, ello se debe a que el dispositivo se desconectó, perdiéndose todos esos mensajes (hay que recordar que en las peticiones asíncronas no se asegura la recepción). Una vez reconectado el maestro, el supervisor sigue recibiendo las tramas correspondientes.

Una última prueba realizada consistió en la realización de una petición asíncrona al nodo 1 por medio de un start (usando como periodo 150ms), para luego realizar un stop, y así sucesivamente.



```

osl1@osl1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
osl1@osl1:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 1
Bulk Frame, Source = 1 Size 2: 0 5
Bulk Frame, Source = 1 Size 2: 1 5
Bulk Frame, Source = 1 Size 2: 2 5
Bulk Frame, Source = 1 Size 2: 3 5
Stop acknowledge frame, Source = 1
Start acknowledge frame, Source = 1
Bulk Frame, Source = 1 Size 2: 4 5
Bulk Frame, Source = 1 Size 2: 5 5
Bulk Frame, Source = 1 Size 2: 6 5
Bulk Frame, Source = 1 Size 2: 7 5
Stop acknowledge frame, Source = 1
Start acknowledge frame, Source = 1
Bulk Frame, Source = 1 Size 2: 8 5
Bulk Frame, Source = 1 Size 2: 9 5
Bulk Frame, Source = 1 Size 2: 10 5

```

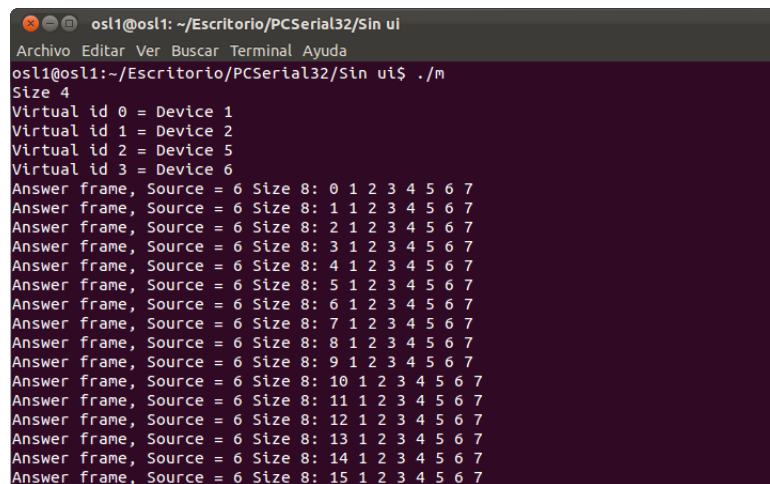
Figura 10.4: Peticiones asíncronas a un nodo común.

Vemos como se van recibiendo los ack correspondientes a las peticiones de start y stop correspondientes.

10.1.3. Petición Síncrona al nodo maestro.

En esta prueba se pretende comprobar la correctitud a la hora de realizar peticiones síncronas desde el supervisor al propio dispositivo maestro.

En la siguiente simulación vemos como al realizar las peticiones al nodo maestro, vamos obteniendo las tramas de respuesta correspondiente. Como se puede observar, el dispositivo maestro al que está conectado el supervisor es el nodo virtual 3 cuya id real es 6.



```
os1@osl1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
os1@osl1:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Answer frame, Source = 6 Size 8: 0 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 1 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 2 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 3 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 4 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 5 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 6 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 7 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 8 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 9 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 10 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 11 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 12 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 13 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 14 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 15 1 2 3 4 5 6 7
```

Figura 10.5: Peticiones síncronas a un nodo maestro.

10.1.4. Petición Asíncrona a un nodo maestro.

En esta prueba se pretende comprobar la correctitud a la hora de realizar peticiones asíncronas desde el supervisor al propio dispositivo maestro.

Para realizar la prueba, se realizó un start (usando como periodo 150ms) al maestro monitorizándose pues los datos recibidos.


```

os11@os11: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
os11@os11:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 6
Bulk Frame, Source = 6 Size 2: 0 6
Bulk Frame, Source = 6 Size 2: 1 6
Bulk Frame, Source = 6 Size 2: 2 6
Bulk Frame, Source = 6 Size 2: 3 6
Bulk Frame, Source = 6 Size 2: 4 6
Bulk Frame, Source = 6 Size 2: 5 6
Bulk Frame, Source = 6 Size 2: 6 6
Bulk Frame, Source = 6 Size 2: 7 6
Bulk Frame, Source = 6 Size 2: 8 6
Bulk Frame, Source = 6 Size 2: 9 6
Bulk Frame, Source = 6 Size 2: 10 6
Bulk Frame, Source = 6 Size 2: 11 6
Bulk Frame, Source = 6 Size 2: 12 6
Bulk Frame, Source = 6 Size 2: 13 6
Bulk Frame, Source = 6 Size 2: 14 6

```

Figura 10.6: Peticiones asíncronas a un nodo común.

10.1.5. Petición Síncrona y Asíncrona a un nodo común.

En esta prueba se pretende comprobar la correctitud a la hora de realizar peticiones síncronas y asíncronas desde el supervisor a diferentes nodos de la red. A la hora de realizar la petición asíncrona correspondiente al start, se utilizó como periodo 120ms.

```

os11@os11: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
os11@os11:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 1
Answer frame, Source = 1 Size 8: 250 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 238 5
Bulk Frame, Source = 1 Size 2: 239 5
Answer frame, Source = 1 Size 8: 251 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 240 5
Bulk Frame, Source = 1 Size 2: 241 5
Bulk Frame, Source = 1 Size 2: 242 5
Answer frame, Source = 1 Size 8: 252 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 243 5
Bulk Frame, Source = 1 Size 2: 244 5
Bulk Frame, Source = 1 Size 2: 245 5
Answer frame, Source = 1 Size 8: 253 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 246 5
Bulk Frame, Source = 1 Size 2: 247 5
Bulk Frame, Source = 1 Size 2: 248 5

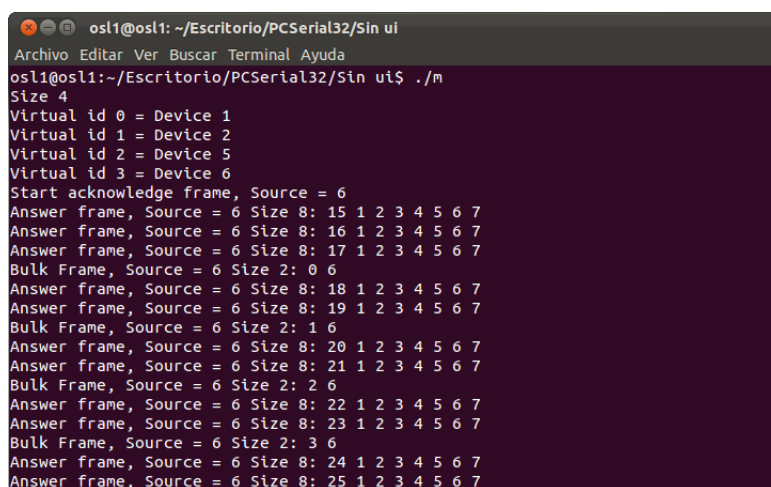
```

Figura 10.7: Peticiones asíncronas a un nodo común.

Como se puede ver en la figura anterior, el nodo supervisor fue recibiendo las respuestas tanto de las peticiones síncronas como de las peticiones asíncronas.

10.1.6. Petición Síncrona y Asíncrona a un nodo maestro.

En esta prueba se pretende comprobar la correctitud a la hora de realizar peticiones síncronas y asíncronas desde el supervisor al propio dispositivo maestro. A la hora de realizar la petición asíncrona correspondiente al start, se utilizó como periodo 120ms.

A terminal window with a dark background and light text. The window title is 'osl1@osl1: ~/Escritorio/PCSerial32/Sin ui'. The menu bar includes 'Archivo', 'Editar', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'. The command prompt shows 'osl1@osl1:~/Escritorio/PCSerial32/Sin ui\$./m'. The output consists of several lines: 'Size 4', 'Virtual id 0 = Device 1', 'Virtual id 1 = Device 2', 'Virtual id 2 = Device 5', 'Virtual id 3 = Device 6', 'Start acknowledge frame, Source = 6', followed by a series of 'Answer frame, Source = 6 Size 8: [sequence of numbers 1-7]' and 'Bulk Frame, Source = 6 Size 2: [number]' pairs, with the sequence of numbers increasing from 15 to 25.

```
osl1@osl1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
osl1@osl1:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 6
Answer frame, Source = 6 Size 8: 15 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 16 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 17 1 2 3 4 5 6 7
Bulk Frame, Source = 6 Size 2: 0 6
Answer frame, Source = 6 Size 8: 18 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 19 1 2 3 4 5 6 7
Bulk Frame, Source = 6 Size 2: 1 6
Answer frame, Source = 6 Size 8: 20 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 21 1 2 3 4 5 6 7
Bulk Frame, Source = 6 Size 2: 2 6
Answer frame, Source = 6 Size 8: 22 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 23 1 2 3 4 5 6 7
Bulk Frame, Source = 6 Size 2: 3 6
Answer frame, Source = 6 Size 8: 24 1 2 3 4 5 6 7
Answer frame, Source = 6 Size 8: 25 1 2 3 4 5 6 7
```

Figura 10.8: Peticiones asíncronas a un nodo común.

Tras realizar dichas peticiones, el nodo maestro responde sin problemas a las peticiones del propio supervisor.

10.1.7. Petición Síncrona y Asíncrona a un nodo común y a un nodo maestro.

En esta prueba se pretende comprobar la correctitud a la hora de realizar peticiones síncronas y asíncronas desde el supervisor a cualquier tipo de nodo.

En la siguiente prueba, se harán peticiones síncronas a un nodo maestro y peticiones asíncronas a un nodo común. A la hora de realizar la petición asíncrona correspondiente al start, se utilizó como periodo 120ms.

```

os11@os11: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
os11@os11:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Answer frame, Source = 6 Size 8: 250 1 2 3 4 5 6 7
Start acknowledge frame, Source = 1
Bulk Frame, Source = 1 Size 2: 0 5
Bulk Frame, Source = 1 Size 2: 1 5
Answer frame, Source = 6 Size 8: 251 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 2 5
Bulk Frame, Source = 1 Size 2: 3 5
Bulk Frame, Source = 1 Size 2: 4 5
Answer frame, Source = 6 Size 8: 252 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 5 5
Bulk Frame, Source = 1 Size 2: 6 5
Answer frame, Source = 6 Size 8: 253 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 7 5
Bulk Frame, Source = 1 Size 2: 8 5
Answer frame, Source = 6 Size 8: 254 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 9 5

```

Figura 10.9: Peticiones asíncronas a un nodo común.

Análogamente a la prueba anterior, en la siguiente prueba se hará en esta ocasión peticiones síncronas a un nodo común y peticiones asíncronas a un nodo maestro.

```

os11@os11: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
os11@os11:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 6
Answer frame, Source = 1 Size 8: 20 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 21 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 22 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 23 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 24 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 25 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 26 1 2 3 4 5 6 7
Bulk Frame, Source = 6 Size 2: 0 6
Answer frame, Source = 1 Size 8: 27 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 28 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 29 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 30 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 31 1 2 3 4 5 6 7
Answer frame, Source = 1 Size 8: 32 1 2 3 4 5 6 7
Bulk Frame, Source = 6 Size 2: 1 6

```

Figura 10.10: Peticiones asíncronas a un nodo común.

10.2. Pruebas de Rendimiento

En este apartado se quiso poner a prueba los valores y tiempos a los cuales la aplicación puede responder.

Tras las pruebas realizadas se pudo comprobar que si se realiza una petición asíncrona a un nodo, si el periodo escogido es menor de 75ms se pueden presentar problemas si al mismo tiempo se realizan peticiones síncronas. A continuación se muestra un ejemplo al usar como periodo 50ms.

```

os1@os1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
os1@os1:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 1
Bulk Frame, Source = 1 Size 2: 170 5
Bulk Frame, Source = 1 Size 2: 172 5
Bulk Frame, Source = 1 Size 2: 173 5
Bulk Frame, Source = 1 Size 2: 174 5
Synchronous command not answered
Retry 1 of Synchronous Command
Bulk Frame, Source = 1 Size 2: 176 5
Bulk Frame, Source = 1 Size 2: 177 5
Bulk Frame, Source = 1 Size 2: 180 5
Bulk Frame, Source = 1 Size 2: 181 5
Bulk Frame, Source = 1 Size 2: 182 5
Bulk Frame, Source = 1 Size 2: 184 5
Synchronous command not answered

```

Figura 10.11: Peticiones asíncronas a un nodo común con periodo de 50ms.

En la figura anterior se aprecia como se van recibiendo mensajes de tipo bulk frame sin embargo no se recibe respuesta a las peticiones síncronas. La razón se debe principalmente a que el dispositivo maestro está procesando muchos datos debido a la petición asíncrona (está recibiendo de forma constante paquetes de tipo bulks), de forma que no le da tiempo a contestar a tiempo a la petición síncrona, ello provoca que se produzcan timeouts asociados a la petición síncrona.

En la siguiente figura se puede ver el resultado obtenido tras repetir la misma prueba pero en esta ocasión tomando como periodo 75ms.

```

os1@os1: ~/Escritorio/PCSerial32/Sin ui
Archivo Editar Ver Buscar Terminal Ayuda
os1@os1:~/Escritorio/PCSerial32/Sin ui$ ./m
Size 4
Virtual id 0 = Device 1
Virtual id 1 = Device 2
Virtual id 2 = Device 5
Virtual id 3 = Device 6
Start acknowledge frame, Source = 1
Bulk Frame, Source = 1 Size 2: 0 5
Answer frame, Source = 1 Size 8: 75 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 1 5
Bulk Frame, Source = 1 Size 2: 2 5
Bulk Frame, Source = 1 Size 2: 3 5
Bulk Frame, Source = 1 Size 2: 4 5
Bulk Frame, Source = 1 Size 2: 5 5
Answer frame, Source = 1 Size 8: 76 1 2 3 4 5 6 7
Bulk Frame, Source = 1 Size 2: 6 5
Bulk Frame, Source = 1 Size 2: 7 5
Bulk Frame, Source = 1 Size 2: 8 5
Bulk Frame, Source = 1 Size 2: 9 5
Bulk Frame, Source = 1 Size 2: 10 5
Bulk Frame, Source = 1 Size 2: 11 5
Bulk Frame, Source = 1 Size 2: 12 5

```

Figura 10.12: Peticiones asíncronas a un nodo común con periodo de 75ms.

Capítulo 11

Conclusiones

La terminación exitosa de este proyecto fin de grado se ha debido en gran medida a la planificación previa en cada una de las etapas. Sin lugar a dudas, la etapa de investigación es aquella a la que más importancia hay que dar, de ella dependerá el rumbo final que tome el proyecto. Como ingenieros, resulta imprescindible tener en cuenta todas las posibilidades a nuestro alcance, aprovechando siempre que se pueda el trabajo realizado por otros profesionales y aportando nuestro granito de arena dando pie a que nuestro trabajo sea de utilidad para la comunidad.

Tras finalizar este TFG que tenía por objetivo la elaboración del lado supervisor en la librería touCAN, se han podido sacar las siguientes conclusiones:

- Realizar una buena gestión de la memoria en arduino resulta vital, ya que hay que tener en cuenta que se ha de procurar proporcionar al usuario de memoria suficiente para que pueda elaborar su programa.
- Distribuir el trabajo realizado bajo una licencia libre, sin duda es la mejor forma de apoyar que el proyecto tenga una continuidad.
- Es importante controlar mediante timeouts posibles pérdidas de paquete, ya de que no ser así, se podría dar el caso de que un dispositivo se quede bloqueado esperando (en caso de ser una petición bloqueante).

Por otra parte, se han cumplido de forma exitosa los objetivos académicos:

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

11.1. Trabajo Futuro

- **Interfáz gráfica:** Una posible continuación respecto a este proyecto podría ser la creación de una interfáz gráfica que permita a usuarios que no tengan conocimientos de programación hacer uso de la librería para monitorizar una red de controladores.
- **Salvado de información:** Dado que en muchas ocasiones el flujo de información a supervisar puede llegar a ser elevado para realizarlo en tiempo real, se podría añadir la funcionalidad de poder guardar en un fichero el resultado de todas las peticiones, permitiendo una depuración offline.
- **Métodos alternativos de comunicación:** Para ciertos proyectos, la necesidad de que el supervisor se conecte por medio de cable al PC puede resultar en un inconveniente. Como alternativas a la comunicación serie del supervisor con la red de controladores, se podría plantear la implementación del supervisor haciendo uso de ondas de radio. Para ello haría falta adquirir un shield que dote al nodo maestro de la capacidad de poder enviar y recibir datos por radio. Del mismo modo habría que adquirir su correspondiente en el PC.
- **Tiempos de respuesta:** Dado que en este tipo de proyectos los tiempos de respuestas toman gran relevancia, se podría trabajar en torno a la mejora de los tiempos tanto en las librerías de los controladores como en la del PC.

Capítulo 12

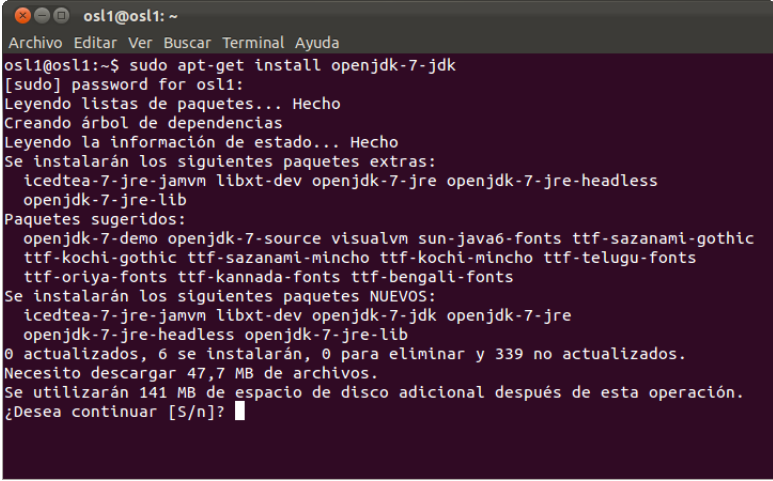
Manual de Usuario

12.1. Instalación

12.1.1. Lado Controlador

Para la instalación de la librerías en el lado controlador, deberemos tener instalado previamente el entorno de compilación de arduino. A continuación se describen los pasos para la correcta instalación en ubuntu 12.04:

1. **Instalación de OpenJDK7:** Abrimos un terminal y ejecutamos el siguiente comando, `sudo apt-get install openjdk-7-jdk` e introduciendo "s" para confirmar la instalación del mismo.



```
os1@os1: ~
Archivo Editar Ver Buscar Terminal Ayuda
os1@os1:~$ sudo apt-get install openjdk-7-jdk
[sudo] password for os1:
Leyendo listas de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes extras:
 icedtea-7-jre-jamvm libxt-dev openjdk-7-jre openjdk-7-jre-headless
  openjdk-7-jre-lib
Paquetes sugeridos:
  openjdk-7-demo openjdk-7-source visualvm sun-java6-fonts ttf-sazanami-gothic
  ttf-kochi-gothic ttf-sazanami-mincho ttf-kochi-mincho ttf-telugu-fonts
  ttf-oriya-fonts ttf-kannada-fonts ttf-bengali-fonts
Se instalarán los siguientes paquetes NUEVOS:
  icedtea-7-jre-jamvm libxt-dev openjdk-7-jdk openjdk-7-jre
  openjdk-7-jre-headless openjdk-7-jre-lib
0 actualizados, 6 se instalarán, 0 para eliminar y 339 no actualizados.
Necesito descargar 47,7 MB de archivos.
Se utilizarán 141 MB de espacio de disco adicional después de esta operación.
¿Desea continuar [S/n]?
```

Figura 12.1: Instalación de openjdk 7.

2. **Instalación del compilador AVR:** Tras instalar jdk, procedemos a instalar el

compilador necesario por medio del siguiente comando, `sudo apt-get install gcc-avr avr-libc binutils-avr avrdude`



```
os1@os1:~  
Archivo Editar Ver Buscar Terminal Ayuda  
os1@os1:~$ sudo apt-get install gcc-avr avr-libc binutils-avr avrdude  
Leyendo listas de paquetes... Hecho  
Creando árbol de dependencias  
Leyendo la información de estado... Hecho  
Paquetes sugeridos:  
  avrdude-doc task-c-devel gcc-doc gcc-4.2  
Se instalarán los siguientes paquetes NUEVOS:  
  avr-libc avrdude binutils-avr gcc-avr  
0 actualizados, 4 se instalarán, 0 para eliminar y 339 no actualizados.  
Necesito descargar 16,9 MB de archivos.  
Se utilizarán 68,8 MB de espacio de disco adicional después de esta operación.  
0% [Esperando las cabeceras]
```

Figura 12.2: Instalación de avr.

3. **Descargar el entorno de programación:** El siguiente paso consiste en bajarse el IDE que se usará para editar y grabar los programas en arduino. Para ello tenemos dos opciones:
 - a) **Centro de Software:** Realizando la búsqueda de Arduino en el centro de software, nos dará la posibilidad de instalar el IDE.
 - b) **Página oficial:** Accediendo a la web oficial de arduino <http://arduino.cc/es/Main/Software> y descargar la versión correspondiente para Linux. Una vez descargado, se procede a descomprimir el paquete en el directorio de instalación deseado.

Una vez instalado el entorno, se deberá de acceder a la carpeta de librerías correspondientes al microcontrolador y copiar la carpeta TouCAN y MS2Timer (librería requerida por TouCAN) en la carpeta "libraries" del directorio donde se haya instalado arduino. Tras copiar la carpeta, ya se podrá hacer uso de los ejemplos proporcionados y grabarlos en el controlador.

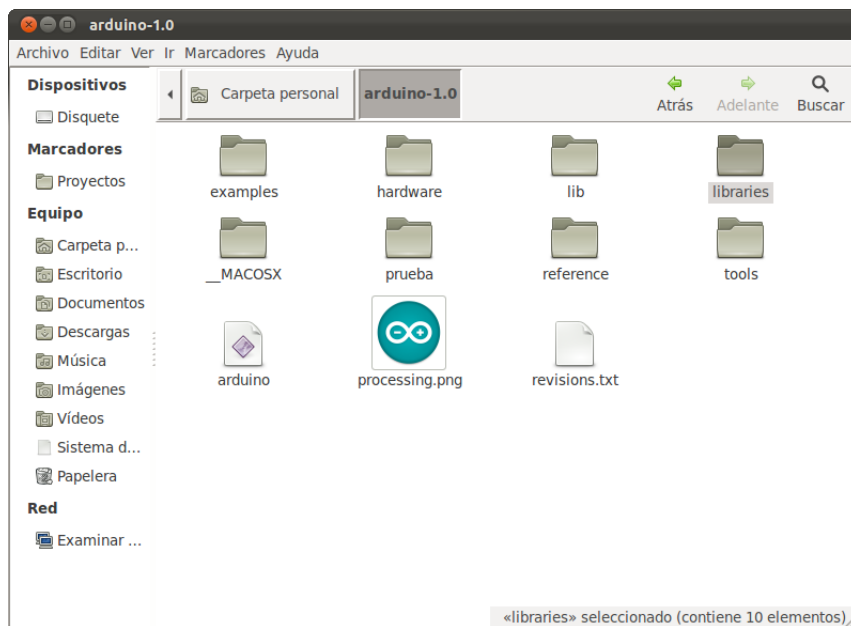


Figura 12.3: Carpeta de librerías.

12.1.2. Lado Supervisor

Como requisito para poder utilizar la librería touCAN correspondiente del supervisor, deberemos tener instalado un compilador de c++. Para su instalación en ubuntu 12.04 se deberá ejecutar `sudo apt-get install g++` en el terminal.

```
os11@os11: ~
Archivo Editar Ver Buscar Terminal Ayuda
os11@os11:~$ sudo apt-get install g++
Leyendo listas de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Paquetes sugeridos:
  g++-multilib
Se instalarán los siguientes paquetes NUEVOS:
  g++
0 actualizados, 1 se instalarán, 0 para eliminar y 339 no actualizados.
Necesito descargar 1434 B de archivos.
Se utilizarán 41,0 kB de espacio de disco adicional después de esta operación.
0% [Esperando las cabeceras]
```

Figura 12.4: Instalación de g++.

12.2. Compilación y Ejecución

12.2.1. Lado Controlador

Junto con la librería TouCAN para los controladores, se aportan una serie de programas de ejemplos desarrollados para grabar en los controladores.

- *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.

- *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.

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

Para grabar uno de estos ejemplos, deberemos de abrirlo con el IDE de arduino y tras realizar los cambios convenientes, compilarlo y grabarlo en la placa. A continuación se muestran capturas del proceso. En concreto se grabará el código correspondiente a un maestro supervisor.

Para la apertura del ejemplo podremos hacerlo directamente arrastrando el fichero a la ventana de edición del IDE, o por medio del menú de apertura destinado a tal fin.



Figura 12.5: Apertura y edición de un programa.

Como vemos, dentro de la ventana de edición podremos realizar los cambios. Una vez se ha terminado realizar las modificaciones en caso de querer compilar se hará click al icono bajo la flecha.

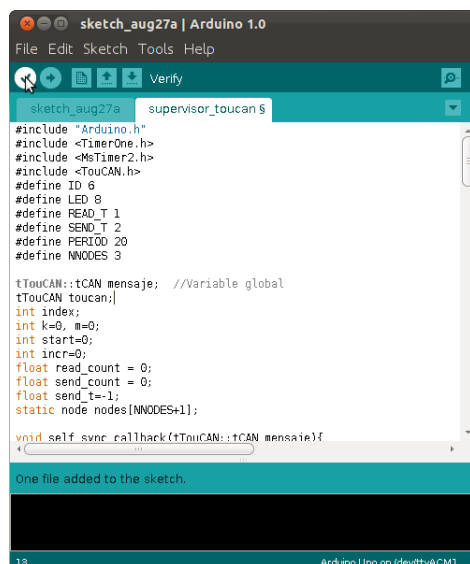


Figura 12.6: Verificación de un programa.

Para poder grabar el programa, antes deberemos seleccionar el dispositivo tal y como se indica en la siguiente figura.

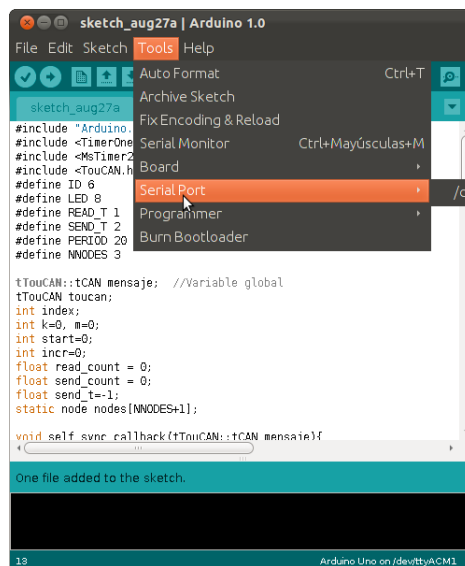


Figura 12.7: Selección del puerto correspondiente al dispositivo.

Una vez seleccionado el dispositivo podremos subir el programa al controlador de la siguiente manera.

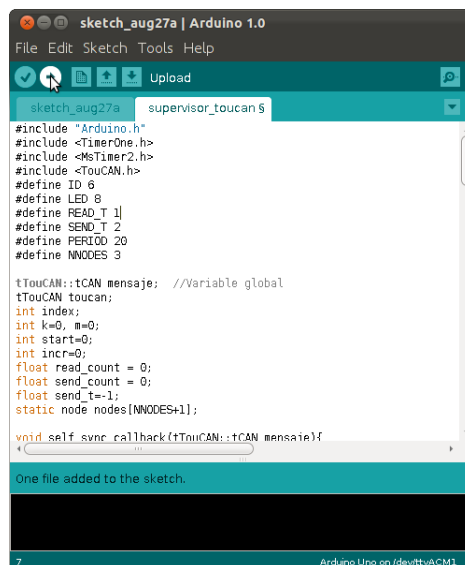


Figura 12.8: Grabación de un programa.

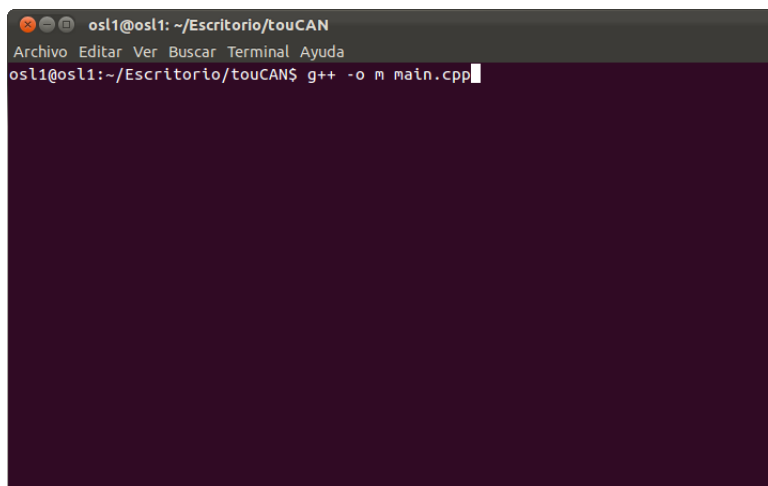
12.2.2. Lado Supervisor

Para la compilación y ejecución en el lado del supervisor, lo primero que deberemos hacer es copiar la carpeta de librería correspondiente al supervisor (dentro de la carpeta Supervisor). Una vez se ha accedido a ella a través del terminal, deberemos

compilar el programa que hace uso de la librería TouCAN. Dentro de la carpeta se proporcionan dos programas de ejemplo.

- *synchronous_example.cpp*: Ejemplo en el cual se realiza una petición síncrona a un dispositivo.
- *asynchronous_example.cpp*: Ejemplo en el cual se realiza una petición asíncrona a un dispositivo.

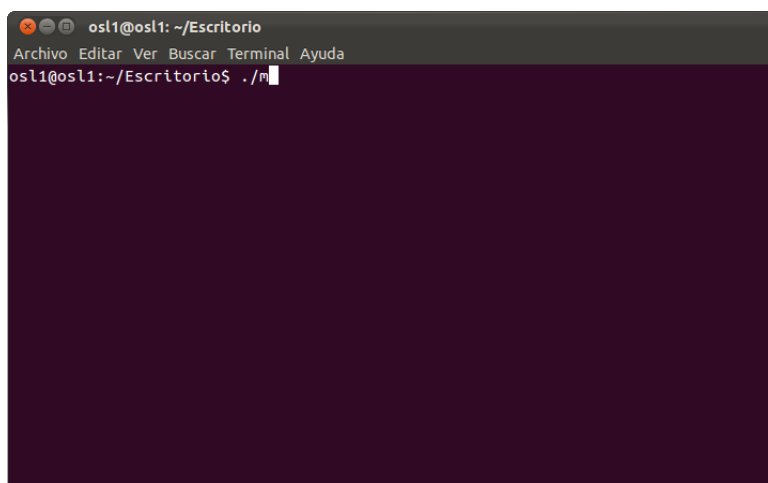
Si quisiéramos compilar deberíamos ejecutar el siguiente comando, `g++ -o main main.cpp` sustituyendo "main" por el nombre del programa a compilar.



```
osl1@osl1: ~/Escritorio/touCAN
Archivo Editar Ver Buscar Terminal Ayuda
osl1@osl1:~/Escritorio/touCAN$ g++ -o m main.cpp
```

Figura 12.9: Compilación del programa supervisor.

Una vez compilado tan sólo resta conectar el supervisor al PC y ejecutar el programa.



```
osl1@osl1: ~/Escritorio
Archivo Editar Ver Buscar Terminal Ayuda
osl1@osl1:~/Escritorio$ ./m
```

Figura 12.10: Ejecución del programa supervisor.

Bibliografía

- [1] Wikipedia, "Can bus." http://en.wikipedia.org/wiki/CAN_bus. [Online; accedida 02-Septiembre-2012].
- [2] Wikipedia, "Protocolo de comunicaciones." http://es.wikipedia.org/wiki/Protocolo_de_comunicaciones. [Online; accedida 02-Septiembre-2012].
- [3] Wikipedia, "Red en bus." http://es.wikipedia.org/wiki/Red_en_bus. [Online; accedida 02-Septiembre-2012].
- [4] R. B. GmbH, "Can specification, version 2.0." http://www.gaw.ru/data/Interface/CAN_BUS.PDF, 1991. [Online; accedida 02-Septiembre-2012].
- [5] Wikipedia, "Microcontrolador." <http://es.wikipedia.org/wiki/Microcontrolador>. [Online; accedida 02-Septiembre-2012].
- [6] "Transceptor." <http://es.wikipedia.org/wiki/Transceptor#Inform.C3.A1tica>. [Online; accedida 02-Septiembre-2012].
- [7] Wikipedia, "Arduino." <http://es.wikipedia.org/wiki/Arduino>. [Online; accedida 02-Septiembre-2012].
- [8] "Arduino uno." <http://arduino.cc/en/Main/ArduinoBoardUno>. [Online; accedida 02-Septiembre-2012].
- [9] "Arduino shields." <http://arduino.cc/en/Main/ArduinoShields>. [Online; accedida 02-Septiembre-2012].
- [10] Sparkfun, "Can-bus shield." <https://www.sparkfun.com/products/10039>. [Online; accedida 02-Septiembre-2012].

- [11] Microchip, "Stand-alone can controller with spi interface." <http://ww1.microchip.com/downloads/en/DeviceDoc/21801G.pdf>, 2012. [Online; accedida 02-Septiembre-2012].
- [12] "Atmega 328." <http://www.atmel.com/devices/atmega328.aspx>. [Online; accedida 02-Septiembre-2012].
- [13] R. M. Pérez, "Arcan." <http://www.arcan.es/>. [Online; accedida 02-Septiembre-2012].
- [14] "Arduino serial." <http://arduino.cc/en/Reference/serial>. [Online; accedida 02-Septiembre-2012].
- [15] "Arduino libraries." <http://arduino.cc/es/Reference/Libraries>. [Online; accedida 02-Septiembre-2012].
- [16] T. E. Kurt, "Arduino-serial: C code to talk to arduino." <http://todbot.com/blog/2006/12/06/arduino-serial-c-code-to-talk-to-arduino/>. [Online; accedida 02-Septiembre-2012].