Trabajo de Fin de Grado

Sistema distribuido de control utilizando buses industriales.

Distributed control system using industrial buses.

Lorenzo Ismael Martín Álvarez

La Laguna, 2 de septiembre de 2016

D. **Alberto Francisco Hamilton Castro**, con N.I.F. 437.738.84-P profesor Titular de Universidad adscrito al Departamento de Nombre del Departamento de la Universidad de La Laguna, como tutor.

C E R T I F I C A

Que la presente memoria titulada:

"Sistema distribuido de control utilizando buses industriales."

ha sido realizada bajo su dirección por D. **Lorenzo Ismael Martín Álvarez**, con N.I.F. 540.499.73-L.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 2 de septiembre de 2016.

Agradecimientos

En primer lugar debo agradecer la labor de mis padres, por luchar sin parar por darles a sus 3 hijos un futuro y una educación que ellos por la situación en la que se encontraban no pudieron alcanzar.

A mis abuelos, especialmente a mi abuelo Lorenzo, el cual siempre fue mi punto de referencia desde que tengo uso de razón.

A mi pareja, gracias a quien entré en la universidad, además de ser quien me ha apoyado en todo momento aguantando todas mis penas y alegrías durante esta etapa de mi vida.

Por último, a mi profesor y tutor Alberto Hamilton, culpable de que eligiese éste trabajo, debido a que él fue quien me introdujo e hizo que me interesase especialmente por el mundo de los microcontroladores, prestándome además toda la ayuda necesaria para la realización de éste proyecto.

Muchas gracias a todos.

Licencia

Resumen

El objetivo de este documento, es explicar la elaboración de un proyecto en el cual se utilizara un sistema distribuido de control utilizando buses industriales.

Se ha optado por utilizar el bus CAN como bus industrial y el Motorola M68HC12 como microcontrolador implementado en la ADAPT912.

Utilizando lo anteriormente mencionado, el objetivo es desarrollar una librería que reúna todas las funcionalidades que un programador que sabe cómo funciona el bus CAN, pero no las interioridades del MS-CAN, necesitaría para hacer uso del bus. Dicha librerías debería permitir configuración de parámetros y velocidades, envío, recepción, uso de interrupciones, etc.

Una vez realizada la librería, se realiza una pequeña aplicación que demuestre el funcionamiento de la librería anteriormente mencionada, utilizando por ejemplo un Display LCD.

Palabras clave: Microcontrolador, M68HC12, Adapt912, Bus CAN.

Abstract

The objective of this document is to explain a project using a distributed control system using an industrial bus.

We choose the CAN bus as industrial bus and the Motorola microcontroller M68HC12 implemented in the ADAPT912.

The main objective was to develop a library that collect all the functionality that a programmer who knows how works the CAN bus, but not the internal work of MS-CAN. The libraries should allow parameters settings, speeds settings, sending, receiving, interrupts, etc.

Once the library was done, the next step is to do an application that shows how works the library, for example using the LCD Display.

Keywords : Microcontroller, M68HC12, Adapt912, CAN Bus.

Índice General

Índice de figuras.

Índice de tablas

Capítulo 1.
Introducción

El proyecto “Sistema distribuido utilizando buses industriales”, se basa en realizar una librería que permita utilizar en un sistema distribuido buses industriales. El sistema distribuido en sí, va a ser simulado con una tarjeta ADAPT912, la cual contiene un microcontrolador M68HC12 y pines destinados al uso del bus utilizado. Para el caso del bus industrial, utilizaremos el bus CAN, un bus ampliamente conocido y utilizado en el mundo de la automoción.

En primer lugar, a pesar de tener ciertos conocimientos sobre el M68HC12, se deben adquirir los conocimientos necesarios sobre el funcionamiento y las particularidades del bus así como ampliar los conocimientos sobre el microcontrolador.

1.1 Antecedentes

CAN es un protocolo de comunicaciones basado en topología bus, desarrollado por Bosch, ampliamente utilizado sobre todo en la industria automovilística. El bus CAN fue elegido debido a que no se disponía de información acerca de desarrollos en el microcontrolador M68HC12 utilizando éste protocolo de comunicaciones, a pesar de que en la documentación del microcontrolador se encuentra bastante información sobre posibles casos de uso.

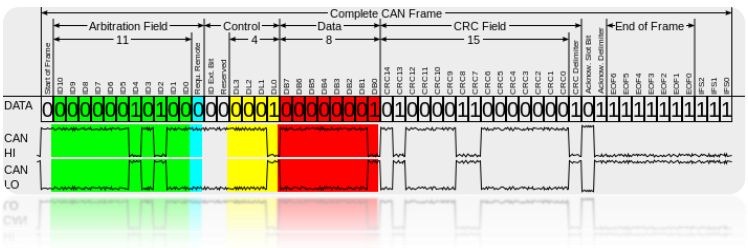


Ilustración 1. Trama CAN

El microcontrolador **M68HC12**, fue introducido en el año 1990 por Motorola, con una CPU de 16 bits. Es el microcontrolador elegido para la realización del proyecto, el cual se encuentra en la placa ADAPT912, debido a que se encuentra preparado para utilizar el protocolo CAN, con una amplia variedad de registros específicos para ello, los cuales se encuentran bien explicados en el manual de uso.

Ilustración 2. M68HC12

ADAPT912, fue diseñado como evaluación y herramienta para el Motorola M68HC912B32. Esta placa, contiene el Motorola M68HC12, además de la conexión serial utilizada para su programación, pines habilitados para el protocolo CAN, mediante los cuales conectaremos la placa al bus y diferentes pines utilizados para el desarrollo del trabajo.

Ilustración 3. ADAPT912

Can4linux es un driver basado en CAN para Linux, el cual fue desarrollado a mediados de los 90. Sobre éste se desarrollan diferentes librerías que serán utilizadas durante el proyecto.

CPC_PCI es la placa utilizada en el proyecto, la cual sirve como interfaz entre el pc y el bus CAN.

1.2 Objetivos.

En principio, como primer objetivo, se marca la familiarización con el bus CAN, sus diferentes tipos de tramas, sus singularidades con respecto al tiempo de envío, parametrización, etc. A pesar de tener algunos conocimientos básicos sobre el microcontrolador M68HC12, la parte relativa al protocolo CAN debe ser estudiada, como por ejemplo los diferentes registros existentes para el bus CAN, parámetros, librerías necesarias, control de tiempos, frecuencias, etc.

Sin embargo, como objetivo final, se marca el desarrollo de una librería escrita en C, implementada en un M68HC12, debido a que a pesar de que existe un periferico CAN, no existe ninguna librería que permita manejar dicho periferico.

Por esto, la librería debería reunir todas las funcionalidades que un programador que sabe cómo funciona el bus CAN, pero no las interioridades del MS-CAN, necesitaría para hacer uso del bus. Dicha librería debería permitir configuración de parámetros y velocidades, envío, recepción, uso de interrupciones, etc.

Capítulo 2.
Estudio previo.

Antes de comenzar con el desarrollo de la librería, se debe conocer a fondo todos los componentes que participan en el proyecto.

2.1 Bus CAN.

- CAN es un protocolo de comunicaciones desarrollado por la firma alemana Robert Bosch, basado en una topología bus para la transmisión de mensajes en entornos distribuidos.
- El protocolo de comunicaciones CAN proporciona los siguientes beneficios:
  - Ofrece alta inmunidad a las interferencias, habilidad para el autodiagnóstico y la reparación de errores de datos.
  - Es un protocolo de comunicaciones normalizado, con lo que se simplifica y economiza la tarea de comunicar subsistemas de diferentes fabricantes sobre una red común o bus.
  - El procesador anfitrión (host) delega la carga de comunicaciones a un periférico inteligente, por lo tanto el procesador anfitrión dispone de mayor tiempo para ejecutar sus propias tareas.
  - Al ser una red multiplexada, reduce considerablemente el cableado y elimina las conexiones punto a punto, excepto en los enganches.
- Las principales características del Bus CAN son:
  - Prioridad de mensajes.
  - Garantía de tiempos de latencia.
  - Sistema de difusión: Los mensajes no indican un destinatario en concreto, sino el tipo de información que está enviando.
  - Detección y señalización de errores.

- Existen dos tipos de bus CAN:
  - CAN de alta velocidad (hasta 1Mbit/s), para partes del sistema que sean críticas.
  - CAN de baja velocidad (hasta 125kbit/s), el cual es tolerante a fallos.

- Capa física.
  - Niveles de tensión del bus:
    - La transmisión de señales en un bus CAN se lleva a cabo a través de dos cables trenzados cuyas señales se denominan CAN_H (CAN high) y CAN_L (CAN low) respectivamente. El bus tiene dos estados definidos: estado dominante y estado recesivo. En estado recesivo, los dos cables del bus se encuentran al mismo nivel de tensión (common-mode voltage), mientras que en estado dominante hay una diferencia de tensión entre CAN_H y CAN_L de al menos 1,5 V.

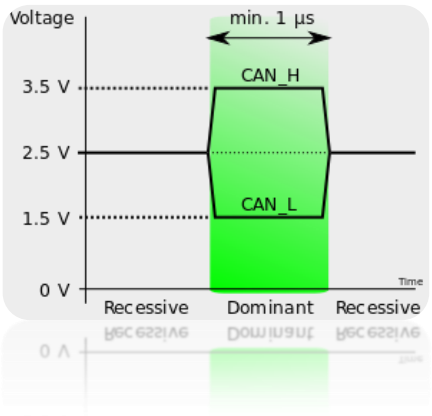


Ilustración 4. Voltaje CAN

  - Sincronización del bus:
    - Todos los nodos de un bus CAN deben trabajar con la misma tasa de transferencia nominal. Por ello es necesario un método de sincronización entre los nodos. En el desarrollo del proyecto, se comprobará como se utiliza una variable llamada prescaler para sincronizar los relojes entre el M68HC12 y el PC.

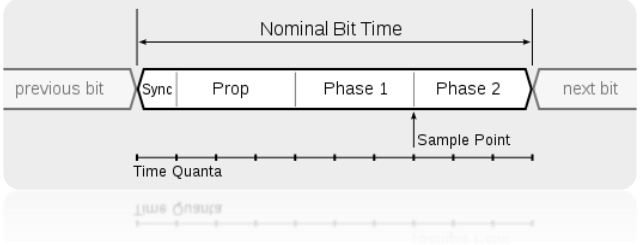


Ilustración 5. Time Quanta

2.2 M68HC12.

El modelo MC68HC912D60A que vamos a utilizar, es un dispositivo de 16 bits. Además, incluye 60KBytes de memoria flash EEPROM, 2Kbyte de memoria RAM, 1Kbytes de memoria EEPROM, dos interfaces de comunicación asíncrona (SCI), conversores analógico-digitales, cuatro canales PWM y un módulo hardware denominado msCAN12, compatible con el protocolo CAN 2.0, el cual contiene:

- 2 buffers de recepción y 3 buffers de emisión.
- Filtros de identificación programables.
- Cuatro canales separados de interrupción, para Rx(recepción), Tx(Trasmisión), Errores y Wake-up(despertar tras modo suspensión).

2.3 ADAPT912.

La tarjeta ADAPT912 fue diseñada como una herramienta para el microcontrolador Mg68hc12. Es una placa ideada para que sea conectada en protoboard sin necesidad de soldaduras. Dicha tarjeta, añade la electrónica necesaria para el cristal del reloj y las diferentes conexiones para los pines.

2.4 Tarjeta CPC_PCI.

- La tarjeta CANPC_PCI es utilizada como interfaz entre el protocolo CAN y el PC, la cual se encuentra equipada con hasta 4 chips SJA1000.

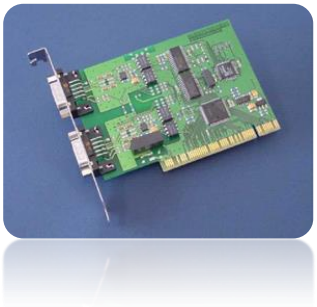


Ilustración 6.
CPC_PCI

2.5 Uso del bus CAN en GNU/Linux.

A parte de lo mencionado anteriormente, para el desarrollo del trabajo son necesarias otras herramientas que son descritas a continuación:

- Configuración de la tarjeta **CPC-CPI** para ser utilizada del mismo modo que una tarjeta de red. En primer lugar, debemos establecer el bitrate y levantar la tarjeta.
  - ip link set can0 type can bitrate 125000 triple-sampling on
  - ifconfig can0 up.
  - Ip details link show can0 (comprobamos su estado).

- Librería can4linux.h, la cual está compuesta por los ficheros can.h y can/raw.h.

- Librerías para compilar código en C para el M68HC12.
  - dpkg -i binutils-m68hc1x_2.18-3.2_i386.deb
  - dpkg -i gcc-m68hc1x_3.3.6+3.1+dfsg-3ubuntu1_i386.deb

2.6 msCan12, el bus CAN en el M68HC12.

El msCan12 utiliza dos pines externos, un pin de entrada (RxCAN) y un pin de salida (TxCAN).

El TxCAN representa el nivel lógico del dispositivo conectado al bus CAN, siendo 0 para un estado dominante y 1 para estado recesivo.

Los 6 pines restantes del puerto CAN están controlados por registros del espacio de direcciones del msCan12, PCTLCAN (puerto de control de registros) y DDRCAN (puerto de dirección de registros).

En la siguiente figura, se muestra un sistema típico utilizando msCan12.

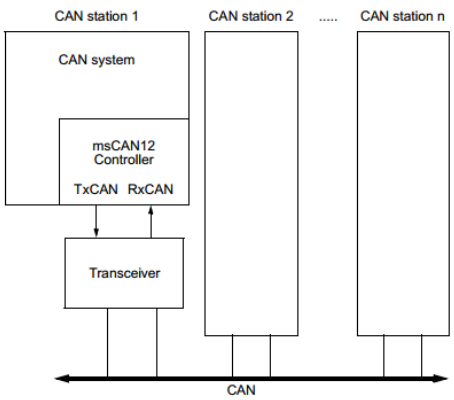


Ilustración 7.
Sistema msCAN12

2.6.1 Almacenamiento de mensajes.

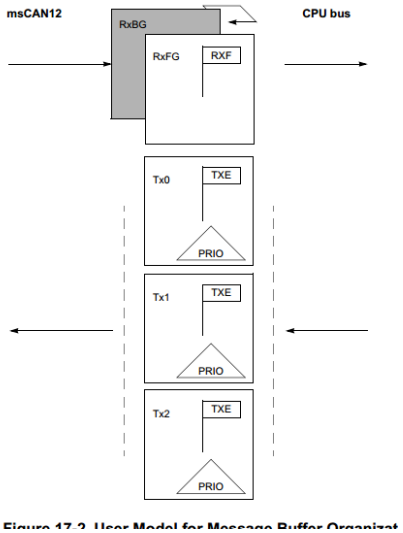
msCan12 facilita un sofisticado almacenamiento de mensajes dirigido a un amplio rango de aplicaciones de red.

- Recepción de mensajes.

  Los mensajes recibidos se almacenan en un FIFO de entrada de dos estapas. Los dos buferrs de mensaje están mapeados en una área de memoria simple. Mientras el backgroundReceiveBuffer (RxBG) esta exclusivamente asociado al msCan12, el ForegroundReceiveBuffer (RxFG) es direccionable desde la CPU12.

  - La mayoría de buffers tiene un tamaño de 13 bytes para almacenar los bits de control de CAN, identificadores y el almacenamiento de datos.
  - The REceiverFullFlag (RXF) está en el registro ReceiverFlagRegister (CRFLG), el cual señala el estado de un buffer de recepción. Cuando el buffer contiene un mensaje recibido correctamente este flag se activa.
  - Además, en cada recepción el mensaje pasa unos filtros de aceptación, donde en el caso de pasar dichos filtros es escrito en el RxBG. A continuación el msCAN12 copia el contenido del RxBG en el RxFG, establece el flag RXF y genera una interrupción a la CPU. Una vez completado esto, el RxBG se encuentra disponible para recibir un nuevo mensaje.

Ilustración 8.
Organización de buffers
de mensajes.



- Cuando el módulo msCAN12 está trasmitiendo, éste recibe su propio mensaje en el RxBG, pero no sobrescribe el RxFG, genera una interrupción de recepción o reconoce su propio mensaje en el bus CAN.

- Estructuras de trasmisión.
  - El msCAN12 tiene un esquema de con tres buffers de trasmisión, para permitir el almacenamientos de multiples mensajes para ser enviados. Los tres buffers contienen una estructura de 13 bytes similar a la estructura de los buffers de recepción.

  - Además, se tiene un buffer de prioridad de registros adicional, llamado TBPR (TransmitBufferPRiorityRegister) el cual contiene 8 bits, para establecer la prioridad entre los buffers de los mensajes a emitir.

  - Para enviar un mensaje de uno de los buffers, la CPU12 debe identificar cual buffer de los 3 está disponible mediante el flag TXE, contenido en el registro CTFLG (TransmitterFlagRegister).

  - Así, la CPU almacena el identificador, los bits de control y el contenido del mensaje en uno de los buffers de trasmisión. Una vez esto se completa, se debe limpiar el buffer TXE, diciendo con esto que el buffer se encuentra listo para ser emitido. La manera de limpiar los buffers en la CPU, es al revés de como se haría normalmente. Por ejemplo, para limpiar el flag TXE, éste debería ponerse a 1.

  - En el caso de que más de un buffer estuviese preparado para ser emitido, el msCan12 utilizaría la configuración de prioridad local.

- Filtros de aceptación.

  - Los registros de aceptación(CIDAR0-7), definen los patrones de aceptabilidad de los identificadores estándar o extendidos (ID10-ID0 o ID28-ID0)
  - Los filtros de aceptación pueden ser programados para operar de 4 modos:

- Dos filtros de aceptación cada uno para ser aplicado con las cabeceras de 29 bits y de 11 bits.
- Cuatro filtros de aplicación.
- Ocho filtros de aceptación.
- Filtros cerrados. Ningún mensaje será copiado en el RxFG y el flag RxF nunca será establecido.

- Interrupciones.

  - El msCAN12, soporta 4 vectores de interrupción, los cuales son:
    - Interrupción de trasmisión.
    - Interrupción de recepción.
    - Interrupción de wake-up (despertar a la cpu después de entrar en modo de reposo).
    - Interrupción de error.

2.6.2 Programación de tiempos del sistema.

La parte de la programación de los tiempos, es bajo mi punto de vista, una de las más complejas del desarrollo del proyecto.
En la siguiente figura se muestra la estructura de la generación del reloj del msCAN12. Con esta flexibilidad del reloj, el msCAN12 puede trabajar con buses CAN con velocidades desde 10kbps hasta 1Mbps.

La fuente de reloj CLKSRC en el registro CMCR1, define si el msCAN12 está conectado a la salida del oscilador de crista(EXTALi) o a un reloj dos veces más rápido que el reloj del sistema (ECLK).

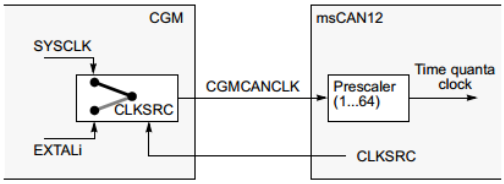


Ilustración 9. Esquema de relojes

- Un prescaler programable se usa para generar desde el msCANCLK el time quanta (Tq). Un time quanta es la unidad atómica manejada por el msCAN12.

$$f_{Tq} = \frac{f_{CGMCANCLK}}{Presc\ value}$$

  - El Time Quanta, vendrá preestablecido por el reloj generado por el M68HC12 y un valor de prescaler que ser obtendrá en función del resto de elementos del bus.
- El tiempo de un bit, esta subdividido en tres segmentos:
  - SYNC_SEG : Este segmento definde el tamaño en un timeQuanta.
  - Time Segment 1: Este segmento puede ser programada en el parámetro TSEG1, el cual puede ir desde 4 – 16 time quanta.
  - Time Segment 2 : Este segmento puede ser programado en el parámetro TSEG2, el cual puede ir desde 2 – 8 time quanta.
- Además el ancho de salto, puede ser programado en un rango de 1 a 4 time quanta, estableciéndolo en el parámetro SJW.

$$BitRate = \frac{f_{Tq}}{number\ of\ TimeQuanta}$$

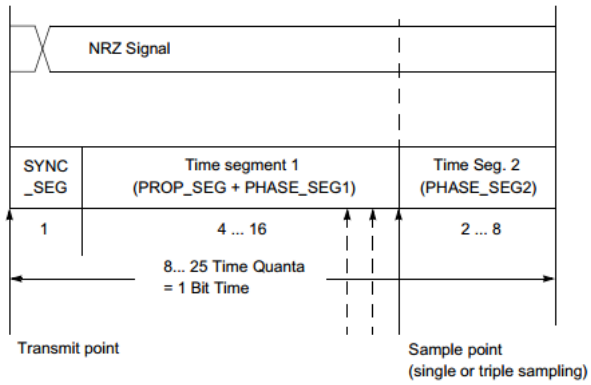


Figure 17-8. Segments within the Bit Time

En la primera figura, podemos ver una gráfica donde se explica el ancho que debe tener el TSG1, TSG2 en función del TimeQuanta y de SYNC_SEG, que 1siempre va a ser igual a 1.

Table 17-3. CAN Standard Compliant Bit Time Segment Settings

Time Segment 1	TSEG1	Time Segment 2	TSEG2	Synchron. Jump Width	SJW
5 .. 10	4 .. 9	2	1	1 .. 2	0 .. 1
4 .. 11	3 .. 10	3	2	1 .. 3	0 .. 2
5 .. 12	4 .. 11	4	3	1 .. 4	0 .. 3
6 .. 13	5 .. 12	5	4	1 .. 4	0 .. 3
7 .. 14	6 .. 13	6	5	1 .. 4	0 .. 3
8 .. 15	7 .. 14	7	6	1 .. 4	0 .. 3
9 .. 16	8 .. 15	8	7	1 .. 4	0 .. 3

En éste caso, se explican las configuraciones del TSEG1, TSEG2 Y SJW.

Ilustración 10. Tq, TSG1, TSG2, SJW

- Mapa de memoria.
  - El msCAN12 ocupa 128bytes en el espacio de memoria de la CPU12. En la siguiente figura se muestra el mapa de memoria del msCAN12.

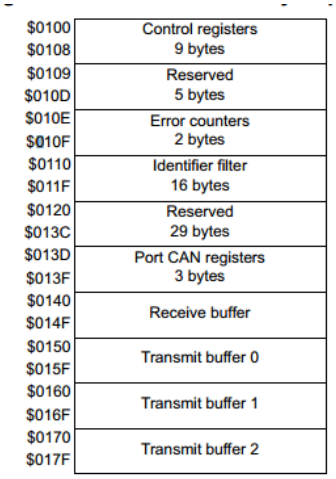


Ilustración 11. Espacio de memoria de msCAN12

- Almacenamiento de mensajes para el programador.
  - A continuación se detalla la organización de los registros de los buffers de trasmisión y de recepción. Por motivos de simplificación para el programador, ambos buffers tienen el mismo estilo. Cada buffer de mensaje ocupa 16 bytes de memoria de los cuales 13 son para la estructura de datos.
  - El registro TBPR solo se encuentra disponible para los buffers de trasmisión.

Address(1)	Register name
01x0	Identifier register 0
01x1	Identifier register 1
01x2	Identifier register 2
01x3	Identifier register 3
01x4	Data segment register 0
01x5	Data segment register 1
01x6	Data segment register 2
01x7	Data segment register 3
01x8	Data segment register 4
01x9	Data segment register 5
01xA	Data segment register 6
01xB	Data segment register 7
01xC	Data length register
01xD	Transmit buffer priority register(2)
01xE	Unused
01xF	Unused

1. x is 4, 5, 6, or 7 depending on which buffer RxFG, Tx0, Tx1, or Tx2 respectively.
2. Not applicable for receive buffers

Ilustración 12.Buffers de transmisión y recepción

- Buffer de datos del mensaje.
    - La siguiente tabla muestra la estructura de recepción de mensajes, cuyo tamaño es de 13 bytes para identificadores extendidos.

ADDR(1)	REGISTER	R/W	BIT 7	6	5	4	3	2	1	BIT 0
$01x0	IDR0	R	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
		W								
$01x1	IDR1	R	ID20	ID19	ID18	SRR (1)	IDE (1)	ID17	ID16	ID15
		W								
$01x2	IDR2	R	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
		W								
$01x3	IDR3	R	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR
		W								
$01x4	DSR0	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x5	DSR1	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x6	DSR2	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x7	DSR3	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x8	DSR4	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x9	DSR5	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01xA	DSR6	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01xB	DSR7	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01xC	DLR	R					DLC3	DLC2	DLC1	DLC0
		W								

Ilustración 13.Buffer de datos del mensaje

- Registros.

    - Registros de identificación (IDRn).
        - Existen dos tipos de identificadores:
            - Identificador estándar (11 bits).
            - Identificador extendido (29 bits).
        - SRR: Este bit se utiliza solamente en el formato extendido.
        - IDE: Este flag indica si se va a utilizar un identificador estandar o extendido.
            - 0 = Formato estandar (11 bits).
            - 1 = Formato extendido (29 bits).

    - Registro de tamaño de datos (DLR).
        - Este registro especifica el tamaño de los datos de la trama CAN, el cual puede variar desde 0 – 8 bytes.

- DLC3 – DLC0.

Data length code				Data byte count
DLC3	DLC2	DLC1	DLC0	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8

Ilustración 14. Registro DLR

- Registro de segmento de datos (DSRn).
  - Los ocho registros de datos, contienen los datos para ser trasmitidos o que están siendo recibidos. El número de bytes que están siendo recibidos o trasmitidos se encuentran en el registro DLR.

- Registro de prioridad de los buffers de trasmisión(TBPR).
  - En el se define la prioridad local que ha sido descrita anteriormente.

		BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
TBPR(1)	R	PRIO7	PRIO6	PRIO5	PRIO4	PRIO3	PRIO2	PRIO1	PRIO0
$01xD	W								
RESET		–	–	–	–	–	–	–	–

1. x is 5, 6, or 7 depending on which buffer Tx0, Tx1, or Tx2 respectively.

Ilustración 15. Registro TBPR

- Registros de control del programador.
  - Registro de tiempos del bus (CBTR0).
    - En este bus se va a definir el SJW descrito con anterioridad así como el prescaler tambien descrito.

		Bit 7	6	5	4	3	2	1	Bit 0
CBTR0	R	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
$0102	W								
RESET		0	0	0	0	0	0	0	0

SJW1	SJW0	Synchronization jump width
0	0	1 Tq clock cycle
0	1	2 Tq clock cycles
1	0	3 Tq clock cycles
1	1	4 Tq clock cycles

Table 17-6. Baud rate prescaler

BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	Prescaler value (P)
0	0	0	0	0	0	1
0	0	0	0	0	1	2
0	0	0	0	1	0	3
0	0	0	0	1	1	4
:	:	:	:	:	:	:
:	:	:	:	:	:	:
1	1	1	1	1	1	64

Ilustración 16. Registro CBTR0.

- Registro de tiempos del bus 1(CBTR1).
  - En éste registro se define el número de muestras enviadas por bit.
  - Además, los TSEG1 y TSEG2 ya descritos, se definen tambien en éste registro.

		Bit 7	6	5	4	3	2	1	Bit 0
CBTR1	R	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10
$0103	W								
RESET		0	0	0	0	0	0	0	0

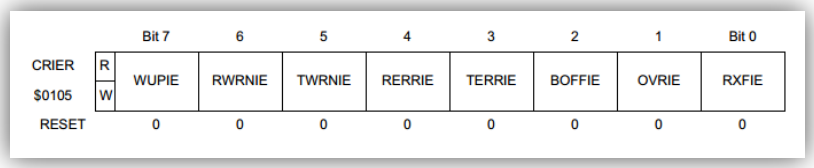
Ilustración 17. Registro CBTR1

- Registro de flags de recepción (CRFLG).
  - En éste registro, el flag mas destacado y al que se le dio uso durante el desarrollo del proyecto fye al flag RxF explicado con anterioridad.

Ilustración 18. Registro CRFLG

- Registro de interrupciones de recepción (CRIER).

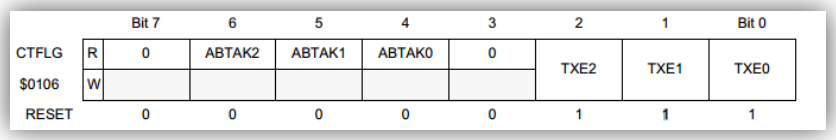
- Al igual que en el registro anterior, el flag destacado es el RXFIE, el cual se activa cuando se genera una interrupción porque un buffer de recepción se encuentra lleno.



		Bit 7	6	5	4	3	2	1	Bit 0
CRIER	R	WUPIE	RWRNIE	TWRNIE	RERRIE	TERRIE	BOFFIE	OVRIE	RXFIE
$0105	W								
RESET		0	0	0	0	0	0	0	0

Ilustración 19. Registro CRIER.

- Registro de flags de trasmisión (CTFLG).
  - En este registro, encontramos los buffers que indican si un buffer de trasmisión se encuentra libre para ser escrito.
  - Cualquiera de los tres flags de trasmisión se pueden limpiar (bajar el flag), escribiendo un 1 en la correspondiente posición.



		Bit 7	6	5	4	3	2	1	Bit 0
CTFLG	R	0	ABTAK2	ABTAK1	ABTAK0	0	TXE2	TXE1	TXE0
$0106	W								
RESET		0	0	0	0	0	1	1	1

Ilustración 20. Registro CTFLG.

Capítulo 3.
Desarrollo del proyecto

En éste apartado, se tratará de explicar todo el desarrollo del proyecto paso a paso, incluyendo el desarrollo progresivo del proyecto, explicaciones sobre funcionalidades, problemáticas encontradas, etc.

3.1 Manejo del bu CAN desde GNU/LINUX.

En primer lugar, debemos descargar el paquete que contiene diferentes utilidades para el bus CAN en GNU/Linux, como son por ejemplo Cansend y Candump, las cuales utilizaremos durante el desarrollo del proyecto para ir probando las funcionalidades del mismo.

Apt-get install can-utils.

- El comando cansend, se utiliza para enviar tramas por el bus CAN en un terminal Linux. Su sintaxis estaría compuesta por:
    - Nombre del dispositivo can por el que se va a enviar (can0, can1…).
    - Una cabecera (de 11 o 29 bits escrita en hexadecimal).
    - Un símbolo divisor entre cabecera y trama "#".
    - Por último, la trama en sí, la cual se escribe también en hexadecimal y está compuesta de 1 – 8 bytes, es decir, de 2 – 16 dígitos (cada número en hexadecimal representa 4 bits).
        - Cansend can0 5B2#11.22.33.44.55.66.77.88.
- El comando candump, se utiliza para ponerse a la escucha de todas las tramas correctas que son enviadas por el bus, sin preocuparse del tamaño de cabecera, filtros de aceptación, etc.

- Para utilizar ambas utilidades utilizando únicamente GNU/Linux, es necesario que haya al menos dos dispositivos en el bus. Para esto, se contaba con dos placas CPC_CPI en la misma máquina, las cuales simulaban dos dispositivos diferentes, puesto que un utilizaba la utilidad cansend y la otra candump.

- Al empezar el proyecto se trató de capturar con un sniffer (wireshark) el tráfico que circulaba por la tarjeta instalada, pero al no haber dos dispositivos no se capturaba ninguna. Una vez añadida un segundo dispositivo que escuchase el tráfico, el sniffer capturaba los paquetes perfectamente.

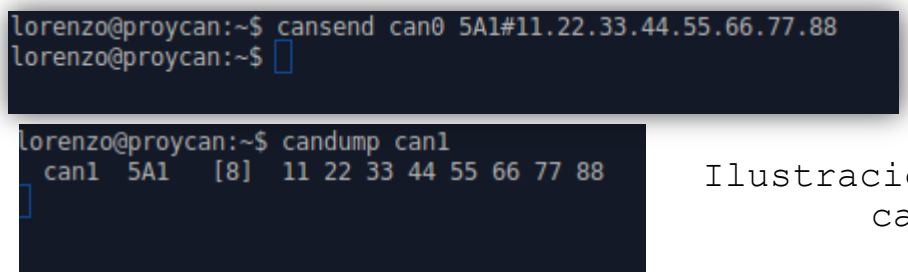


Ilustración 21. Ejemplo canutils

Ejemplo del envío y una recepción de una trama por una tarjeta llamada can0 con la utilidad cansend y la utilidad candump.

A continuación se detallará los pasos dados para la realización de una pequeña aplicación utilizando como lenguaje de programación C y las librerías de can.h para Linux.

3.1.1 Librerías necesarias.

Para el desarrollo de la aplicación que corra en Linux, fueron necesarias varias librerías que detallaré a continuación:

- *Stdio.h*: Cabecera estándar de Entrada/Salida.
- *Unistd.h*: Manejo de directorios y archivos (fclose, fopen,etc.)
- *String.h*: Manejo de strings.
- *Sys/types.h y sys/socket.h:* Librerias para el manejo de sockets.
- *Linux/can.h y Linux/can/raw.h*: Librerias para el manejo del bus can en Linux.

3.1.2 Desarrollo del código.

Al igual que en el protocolo TCP/IP, en primer lugar debemos abrir un socket para comunicarnos en la red CAN. La definición de un socket en C es la siguiente:

- Int socket(intdomain, int type, int protocol).

Por lo que para el caso del bus CAN podría quedar de dos maneras:

- Para el modo RAW, el cual se utilizó:
  - Int socket (PF_CAN , SOCK_RAW , CAN_RAW).
- Para el modo Broadcast:
  - Int socket (PF_CAN , SOCK_DGRAM , CAN_BCM).

Una vez abierto el socket, podríamos realizar diferentes operaciones sobre el, como:

- Read, Write, send, sento, sendmsg.

La estructura de un trama CAN, se encuentra incluida en /Linux/can.h y sería como sigue:

- ```
  struct can_frame {
      canid_t can_id; //32 bits de Cabecera.
  __u8        can_dlc; //Trama (0 .. 8)
      __u8   __pad;   /* padding */
      __u8   __res0;  /* reserved padding */
      __u8   __res1;  /* reserved padding */
      __u8   data[8] __attribute__((aligned(8)));
  };
  ```

- La estructura del socket can es la siguiente:

```
struct sockaddr_can {
    sa_family_t can_family;
   int         can_ifindex;
    union {
    struct { canid_t rx_id, tx_id; } tp;
} can_addr;
 };
```
```

Su funcionamiento sería el siguiente:

- Creamos una estructura:
  - Struct sockaddr_can addr;
  - Struct ifreq ifr;
- Utilizamos la tarjeta can0 y la enlazamos con el socket con la estructura creada.
  - Strcpy(ifr.if_name,"can0").
  - Ioctl(s,SIOCGIFINDEX,&ifr);
  - Addr.can_family = AF_CAN;
  - Addr.can_ifindex = ifr.ifr_ifindex;
  - Bind(s(struct sockaddr*)&addr,sizeof(addr));

- En el caso de que queramos conectar un socket a todas las interfaces CAN, el índice de interfaz debería ser cero, en cuyo caso el socket recibe tramas CAN desde todas las interfaces CAN habilitadas. Para determinar el origen desde el que leer la trama debemos utilizar recvfrom en lugar de read. Así mismo, para enviar desde una interfaz, en lugar de send, deberíamos utilizar sendto.

  - Can_infindex = ZERO;

- Para la lectura de una trama CAN desde un socket CAN_RAW, se utiliza la estructura ya descrita can_frame como sigue:
  - int nbytes.
  - Struct can_frame estructura_frame;
  - Nbytes = read(s,&estructura_frame,sizeof(struct can_frame);

  - Si se utilizase can_infindex = ZERO en lugar de read, utilizaremos recvfrom para obtener el origen del mensaje.
    - En éste caso, obtendremos los datos de origen como sigue:
      - Ifr.if_ifindex = addr.can_ifindex;
      - Iotcl(s,SIOCGIFNAME,&ifr);
      - La interfaz es : ifr.ifr_name.
  - Si nbytes < 0 : O la trama se leyó o no existe ninguna.
  - Si nbytes < sizeof(struct can_frame) : La trama está incompleta.

- En otro caso, utilizar la estructura como se considere.

- Para la escritura de una trama CAN, el funcionamiento sería de forma similar, pero en éste caso rellenaríamos la estructura antes de hacer el envío:
  - Nbytes = write(s,&estructura_frame,sizeof (struct can_frame));
  - Si se utilizase can_infindex = ZERO en lugar de write, utilizaremos sendto para obtener el origen del mensaje.
    - Strcpy(ifr.ifr_name,"can0");
    - Ioctl(s,SIOCGIFINDEX,&ifr);
    - Addr.can_ifindex = ifr.ifr_ifindex;
    - Addr.can_family = AF_CAN;
    - nbytes = sendto(s, &frame, sizeof(struct can_frame), 0, (struct sockaddr*)&addr, sizeof(addr));

- Otras características a tener en cuenta son:
  - Solo se reciben tramas validas (no hay mensajes de error de tramas)
  - La retroalimentacion de las tramas CAN está habilitada:
    - int loopback = 0; /* 0 = disabled, 1 = enabled (default) */
    - setsockopt(s,SOL_CAN_RAW,CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
- Así, su funcionamiento sería el siguiente:

```
while(1){
        nbytes = 0;
   while (nbytes==0){
        nbytes = read(s, &estructura_frame, sizeof(struct can_frame));
   }

   if ( nbytes < 0 || nbytes < sizeof(struct can_frame)){
         printf( " LA TRAMA ESTA VACIA O INCOMPLETA \n" );
         printf( " LA TRAMA ES : \n ");
         printf( " CABECERA : %i ", estructura_frame.can_id);
         exit(0);
        }

   else{
         printf( " LA TRAMA ESTA COMPLETA \n ");
                  printf( " LA TRAMA ES : \n " );
         printf( " CABECERA : %i", estructura_frame.can_id);
         printf( " TRAMA : %u",estructura_frame.can_dlc);
   }
        }
```

- En la segunda imagen se aprecia que los datos de la trama y de la cabecera se encuentran en decimal. Como se comentó con anterioridad, al utilizar valores hexadecimales, los mostraremos como tal sustituyendo %i por %x en el printf.

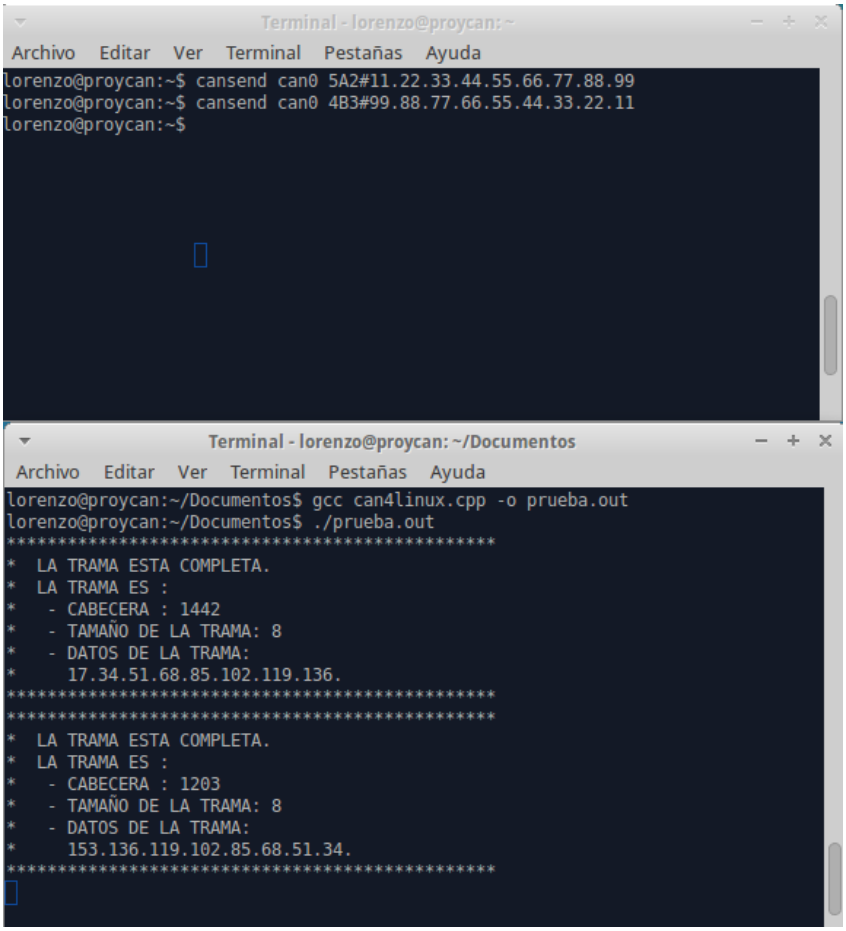


Ilustración 22. Recepcion can.h

3.2 Desarrollo M68HC12.

En éste capítulo se explica lo mas detallado posible el desarrollo de la librería para el M68HC12, poniendo especial énfasis en las cosas donde encontré mayores dificultades.

3.2.1 Librerias y paquetes necesarios.

En primer lugar instalamos los paquetes Debian que proporcionan el ensamblador (GAS), compilador (GCC) y las utilidades que permiten la compilación/ensamblado cruzado para microprocesadores de las familias 68HC11 y 68HC12:

- sudo dpkg -i binutils-m68hc1x_2.18-3.2_i386.deb
- sudo dpkg -i gcc-m68hc1x_3.3.6+3.1+dfsg-3ubuntu1_i386.deb

Las librerías necesarias para el desarrollo son las siguientes:

- #include <sys/interrupts.h>
- #include <sys/sio.h>
- #include <sys/locks.h>

Además, para acceder a los diferentes registros del M68HC12, definiremos la palabra __io_ports[VALOR REGISTRO], como sigue:

- #define _IO_PORTS_W(d) (((unsigned volatile short*) & _io_ports[(d)])[0])

3.2.2 Tiempos.

Éste apartado trata de uno de los aspectos más complejos del trabajo.

- En primer lugar, debemos tener en cuenta el reloj del sistema. En éste caso, el tiempo de un bit se mide en timeQuanta, el cual a su vez se divide en TSYNC, TSEG1 y TSEG2.
- El reloj del adapt va a 8 MHz, es decir un bit tarda en enviarse t = 1/f = ⅛ = 0.125microseg. Por tanto, si elegimos el TQ = 16 (valor parece razonable).
- Mirando con el osciloscopio, en enviar 1 bit con la utilidad cansend se tarda 8microseg, debido a que el bus está configurado a 125kbits/seg.

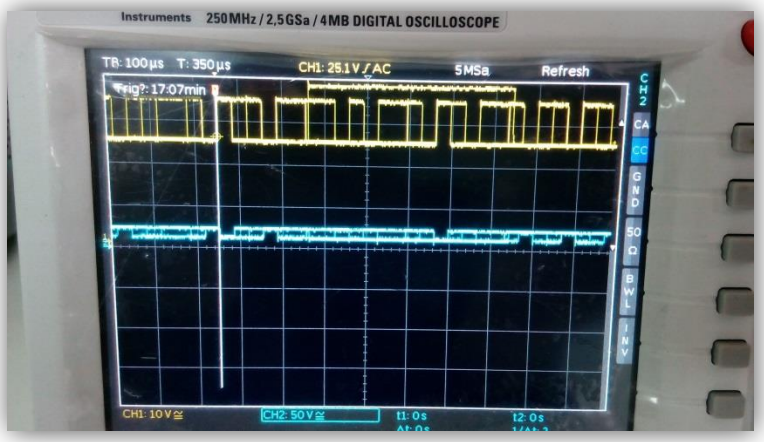


Ilustración 23. Imagen de osciloscopio

- Por tanto, si un bit tarda 8 microseg, y el tq son 16 bits, teniendo que:
    - bitRate = frecTQ / numero de Tq, quedaría:
        - 8(tiempo enviar 1 bit)/16 = 0.5microseg

- Como el bitRate sería igual a 0.5microseg y la máquina(adapt) tarda 0.125microseg en enviar un bit, para que vayan a la misma velocidad, habria que utilizar el prescaler, que sería:

- 0.125 * 4 = 0.5, es decir, habría que dividir entre 4 la velocidad para que vayan igual.

- Así, teniendo en cuenta la ilustración 10, se definen TSEG1 y TSEG2 con respecto al TimeQuanta = 16 y TSYNC = 1,establecemos:
    - 1 < TS1 < 16.
    - 2 < TS2 < 8.

- Así, la elección será la siguiente:
    - SYNC = 1.
    - TSEG1 = 9.
    - TSEG2 = 6.

- Quedando siguiendo el gráfico 1 – 8 – 5.
- Además, para el syncronization jump width, el cual corresponde al número de ciclos que un bit puede retrasarse:
    - SJW = 4. (equivale a 3).

- A continuación, se detallan todos los registros que se necesitan modificar para establecer los tiempos.
    - En el registro **CBTR0**, se establecen el SJW1 y SJW0, cuyo valor se había establecido a 4 en el punto anterior, así como el BAUD RATE PRESCALER, valor de prescaler calculado entre las máquinas anteriormente.

		Bit 7	6	5	4	3	2	1	Bit 0
CBTR0	R	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
$0102	W								
RESET		0	0	0	0	0	0	0	0

Table 17-5. Synchronization jump width

SJW1	SJW0	Synchronization jump width
0	0	1 Tq clock cycle
0	1	2 Tq clock cycles
1	0	3 Tq clock cycles
1	1	4 Tq clock cycles

Table 17-6. Baud rate prescaler

BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	Prescaler value (P)
0	0	0	0	0	0	1
0	0	0	0	0	1	2
0	0	0	0	1	0	3
0	0	0	0	1	1	4
:	:	:	:	:	:	:
:	:	:	:	:	:	:
1	1	1	1	1	1	64

The CBTR0 register can only be written if the SFTRES bit in CMCR0 is set.

Ilustración 24. Tablas CBTR0

- Quedando según las tablas anteriores:
  - SJW1 = 1 y SJW0 = 1.
  - BRPx = 4.

- El siguiente registro a modificar es el **CBTR1**, en el cual se establecen los TSEG1 y TSEG2.

		Bit 7	6	5	4	3	2	1	Bit 0
CBTR1	R	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10
$0103	W								
RESET		0	0	0	0	0	0	0	0

- Quedando según lo descrito:
  - TSEG1 = 9.
  - TSEG0 = 6.

Table 17-8. Time segment values

TSEG13	TSEG12	TSEG11	TSEG10	Time segment 1
0	0	0	0	1 Tq clock cycle
0	0	0	1	2 Tq clock cycles
0	0	1	0	3 Tq clock cycles
0	0	1	1	4 Tq clock cycles
.	.	.	.	.
.	.	.	.	.
1	1	1	1	16 Tq clock cycles

TSEG22	TSEG21	TSEG20	Time segment 2
0	0	0	1 Tq clock cycle
0	0	1	2 Tq clock cycles
.	.	.	.
.	.	.	.
1	1	1	8 Tq clock cycles

- Para modificar los registros CBTR1 y CBTR0, el bit SFTRES en el registro CMCR0, debe estar activado, es decir, el bus debe encontrarse en estado de SOFT_RESET.

		Bit 7	6	5	4	3	2	1	Bit 0
CMCR0	R	0	0	CSWAI	SYNCH	TLNKEN	SLPAK	SLPRQ	SFTRES
$0100	W								
RESET		0	0	1	0	0	0	0	1

Ilustración 25. Registro CMCR0

- El bit 0, debe estar igual a 1 y el resto los dejamos como están:
  - _io_ports[M6812_CMCR0] |= (0 << 1).
- Una vez terminado de inicializar los tiempos, procedemos al envío de datos.

3.2.3 Desarrollo del envío.

Para el envío de datos habrá que tener en cuenta ciertos datos previos:

- La sincronización entre el msCAN12 y el bus.

- Los dos tipos de cabeceras que existe (11 y 29 bits).
- Los tres buffers de trasmisión existentes (Tx0,Tx1 y Tx2).
- Por tanto, se debería sincronizar con el bus, preparar el envío, comproba cuál de los tres buffers está disponible y almacenar en éste el ID, los bits de control y los datos de la trama. Una vez realizado esto, debemos marcar el buffer como preparado para emitir cambiando el flag TxE.

  - Para sincronizar el bus can con el MSCAN12, basta con comprobar si el flag SYNCH, del registro CMCR0 se encuentra activo. Desde el momento que lo esté, se puede considerar que ambos están sincronizados.

  - Para lo descrito anteriormente, comenzaremos comprobando el registro CTFLG, en el cual se indica cuál de los registros de trasmisión se encuentra libre.

		Bit 7	6	5	4	3	2	1	Bit 0
CTFLG	R	0	ABTAK2	ABTAK1	ABTAK0	0	TXE2	TXE1	TXE0
$0106	W								
RESET		0	0	0	0	0	1	1	1

0 = The associated message buffer is full (loaded with a message due for transmission).
1 = The associated message buffer is empty (not scheduled).

Ilustración 26. Registro CTFLG

  - Una vez comprobado cuál de los registros se encuentra libre para ser preparado, la estructura de cualquiera los 3 es la misma, simplemente se debe tener en cuenta la dirección de memoria de cada uno:

ADDR(1)	REGISTER	R/W	BIT 7	6	5	4	3	2	1	BIT 0
$01x0	IDR0	R	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
		W								
$01x1	IDR1	R	ID20	ID19	ID18	SRR (1)	IDE (1)	ID17	ID16	ID15
		W								
$01x2	IDR2	R	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
		W								
$01x3	IDR3	R	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR
		W								
$01x4	DSR0	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x5	DSR1	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x6	DSR2	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x7	DSR3	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x8	DSR4	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01x9	DSR5	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01xA	DSR6	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01xB	DSR7	R	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
		W								
$01xC	DLR	R					DLC3	DLC2	DLC1	DLC0
		W								

Ilustración 27. Estructura buffers

- Dónde:
  - IDRn: 11/29 bits correspondientes a la cabecera.
    - IDE: Indica el tipo de cabecera:
      - IDE = 0 (11 bits).
      - IDE = 1 (29 bits).
  - DLR: Longitud de datos de la trama can (de 0 a 8 bytes).
  - DSRn: Datos a enviar, cuya longitud la determina el DLR.
  - TBPR: Registro de prioridad de los buffers de trasmisión.

- El acceso a los diferentes registros del buffer, se hace accediendo directamente a la posición de memoria de éste. Así, por ejemplo para el buffer Tx0, cuya dirección de memoria está definida como M6812_Tx1 (0x100), para acceder al registro IDR0, bastaría con hacer M6812_Tx1 + 1.

- Un ejemplo de preparación del envío sería:
  - Elegimos por ejemplo, una trama cuya cabecera sea de 11 bits, por lo que el flag IDE debe ser igual a 0 y la cabecera, será 101 1100 0011 = 5 C 3, donde:

- IDR0 = 101 1100 0.
- IDR1 = 011 SRR IDE(=0) 0 0 0.
- IDR2 e IDR3, se quedan sin utilizar.

- Los registros del dsr0 a dsr7, los rellenos con los valores hexadecimales para la trama 11.22.33.44.55.66.77.88
  - DSR0 = 00010001
  - DSR1 = 00100010
  - DSR2 = 00110011
  - DSR3 = 01000100
  - DSR4 = 01010101
  - DSR5 = 01100110
  - DSR6 = 01110111
  - DSR7 = 10001000

- El registro DLR, será = 8 (8bytes de datos), el cual representaremos:
  - 1000 DLC3.

- Una vez tengamos un buffer listo para ser enviado, debemos modificar los flags del registro de transmisión (CTFLG).
- Los registros TXE2, TXE1 y TXE0, indican si el correspondiente buffer al que están asociados se encuentran vacíos. Una vez el mensaje haya sido enviado, la CPU se encarga de limpiar dicho flag, donde:
  - 0 = El buffer asociado está lleno (Listo para transmitir).
  - 1 = El buffer asociado está vacío.

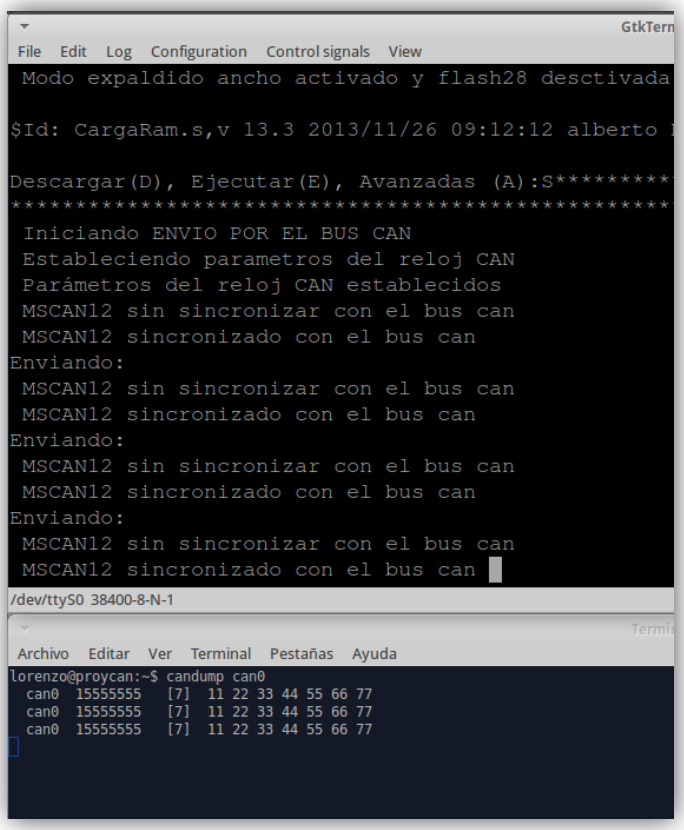


Ilustración 28. Ejemplo envío.

3.2.4 Desarrollo de la recepción.

En principio, el funcionamiento de la recepción será similar al del envío, en cuanto a parámetros de reloj y sincronización se refiere.

Las principales diferencias que nos encontraremos serán:

- Solo existe un buffer de recepción (RxFG) en lugar de tres como ocurre con la emisión, cuya estructura es exactamente igual.
- Utilizaremos modo pooling, que consiste en comprobar continuamente si hay alguna trama disponible y además se utilizará interrupciones para las recepciones, debido a que el envío lo realizamos en el momento que se desea, sin embargo el dispositivo que reciba no sabe en el momento que lo va a hacer.

El modo pooling consiste en mantenernos en un bucle while comprobando continuamente el flag RxF del registro CRFLG hasta que éste se dispare.

RXF — Receive Buffer Full

The RXF flag is set by the msCAN12 when a new message is available in the foreground receive buffer. This flag indicates whether the buffer is loaded with a correctly received message. After the CPU has read that message from the receive buffer, the RXF flag must be handshaken (cleared) in order to release the buffer. A set RXF flag prohibits the exchange of the background receive buffer into the foreground buffer. If not masked, a Receive interrupt is pending while this flag is set.

0 = The receive buffer is released (not full).
1 = The receive buffer is full. A new message is available.

		Bit 7	6	5	4	3	2	1	Bit 0
CRFLG	R	WUPIF	RWRNIF	TWRNIF	RERRIF	TERRIF	BOFFIF	OVRIF	RXF
$0104	W								
RESET		0	0	0	0	0	0	0	0

Ilustración 29. Flag RxF

Una vez dicho flag se dispara, se entiende que existe un mensaje cargado en el buffer RxFG, por lo que pasaremos a interpretarlo de manera similar a los buffers de emisión:

- En primer lugar se debe comprobar el bit IDE correspondiente al registro IDR1. En caso de que éste se encuentre a 1, tendremos una cabecera de 29 bits y en caso de que se encuentre a 0, de 11 bits.
    - El principal incoveniente encontrado con las cabeceras de 29 bits, es que los bits de ésta no se encuentran escritos de manera consecutiva, sino que existen alguno flags intermedios que dificultan la lectura de éstos.

- Una vez interpretada la cabecera, procederemos a anlizar la parte de los datos (DSR0-DSR7) en función del registro DLR que indicará el tamaño de la trama de datos.

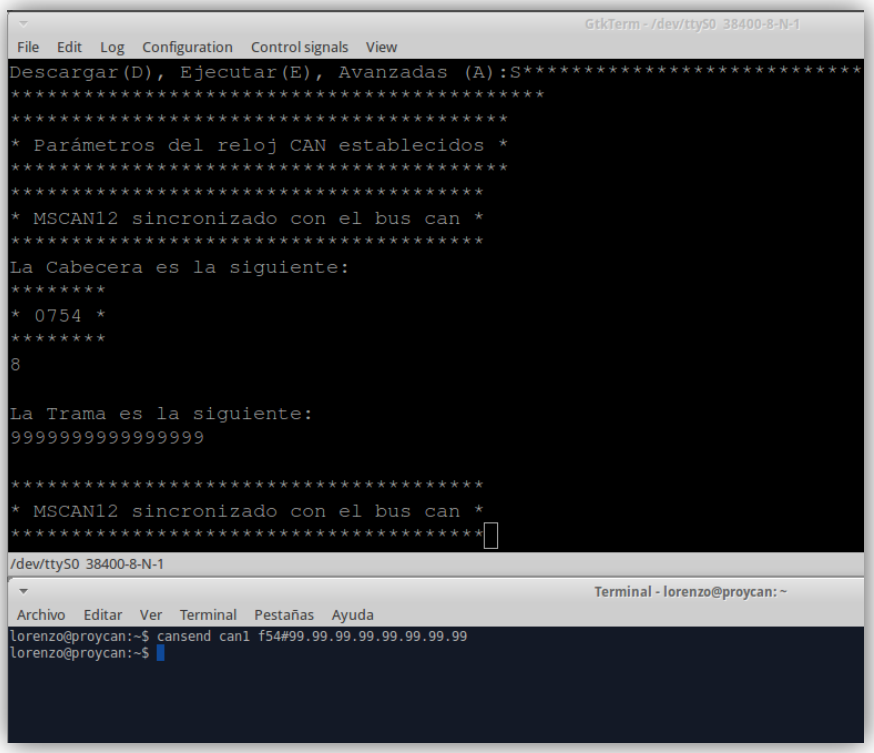


Ilustración 30. Recepciçon con cabecera de 29 bits.

- El siguiente paso a realizar es:
  - En lugar de realizar la recepción en modo pooling que consiste en:
    - Comprobar continuamente si el flag de recepción está activo dentro del propio metodo RecibirCan, lo que origina una recepción bloqueante, ya que el programa se queda en un bucle while comprobando continuamente si el buffer de recepción está lleno(RxF), momento en el que sale del bucle y comienza a realizar la lectura.
  - Lo que se va a realizar es una recepción mediante interrupciones, permitiendo al programa realizar otras funcionalidades cuando no se encuentra recibiendo datos.

- Los registros para manejar las interrupciones del bus CAN son:
  - MSCAN wake-up: CRIE(WUPIE).
  - **MSCAN receive: CRIER(RXFIE).**
  - MSCAN TRANSMIT: CTCR(TXEIE[2:0]).

$FFD0, $FFD1	MSCAN wake-up	I bit	CRIER (WUPIE)	$D0
$FFCE, $FFCF	Key wake-up G or H	I bit	KWIEG[6:0] and KWIEH[7:0]	$CE
$FFCC, $FFCD	Modulus down counter underflow	I bit	MCCTL (MCZI)	$CC
$FFCA, $FFCB	Pulse Accumulator B Overflow	I bit	PBCTL (PBOVI)	$CA
$FFC8, $FFC9	MSCAN errors	I bit	CRIER (RWRNIE, TWRNIE, RERRIE, TERRIE, BOFFIE, OVRIE)	$C8
$FFC6, $FFC7	MSCAN receive	I bit	CRIER (RXFIE)	$C6
$FFC4, $FFC5	MSCAN transmit	I bit	CTCR (TXEIE[2:0])	$C4

Ilustración 31. Registros de interrupción msCAN12

- Para hacer uso de las interrupciones, en primer lugar debemos incluir el fichero sys/interrupts.h.

- La rutina de tratamiento de cada interrupción, se define declarando en el código una función con un nombre predeterminado. Con esto, la librería se encarga de definir el vector necesario en la tabla de vectores de interrupción. Los nombres de las funciones que vamos a utilizar para el caso del bus can, son las siguientes:

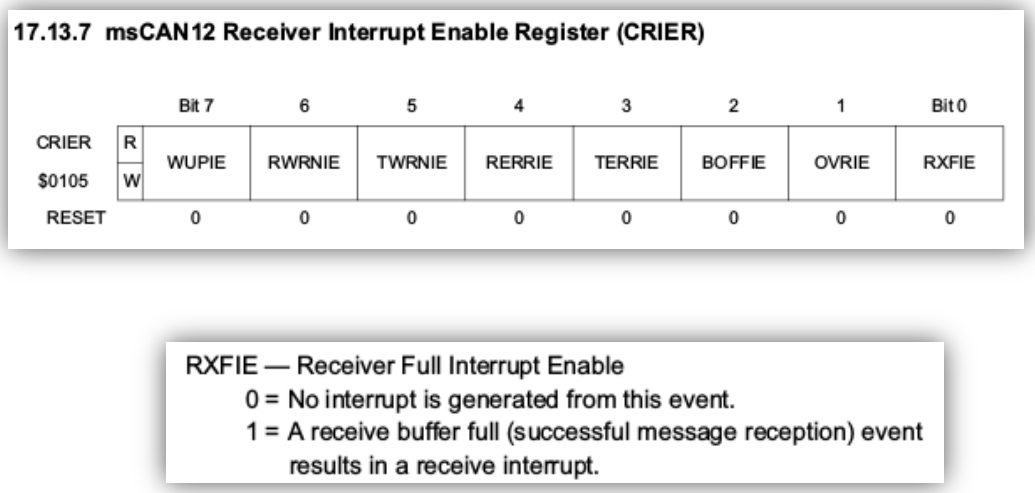
vi_cantx	Transmisión en el subsistema CAN
vi_canrx	Recepción en el subsistema CAN
vi_canerr	Errores en el subsistema CAN

Ilustración 32. Métodos de interrupciones.

- Para el caso que estamos tratando, la recepción, la función quedará como sigue:
  - void __atribute__((interrupt)) vi_canrx (void) { }

- Si en dicha función se debe modificar algo del programa principal, tan solo lo podrá hacer con variables globales.

- Para que dichas interrupciones funcionen, debemos habilitar el flag RXFIE (Receiver full interrupt enable), es decir, habilita una interrupción cuando el buffer se encuentre disponible.



17.13.7 msCAN12 Receiver Interrupt Enable Register (CRIER)

		Bit 7	6	5	4	3	2	1	Bit 0
CRIER	R	WUPIE	RWRNIE	TWRNIE	RERRIE	TERRIE	BOFFIE	OVRIE	RXFIE
$0105	W								
RESET		0	0	0	0	0	0	0	0

RXFIE — Receiver Full Interrupt Enable
0 = No interrupt is generated from this event.
1 = A receive buffer full (successful message reception) event results in a receive interrupt.

Ilustración 33. Registro CRIER.

- Para que el programa no se quede con la interrupción de recepción continuamente habilitada, es decir, accediendo sin parar al metodo de la interrupción (vi_canrx), al terminar de leer el buffer, debemos limpiar el bit RXF (Receiver buffer full), comunicando a la máquina que hemos terminado de leer dicho buffer y que ya éste no se encuentra listo para la lectura.
  - _io_ports[M6812_CRFLG] = M6812B_RXF;

3.3 Preparación de las librerías

Una vez finalizadas las funcionalidades previstas en el desarrollo, pasamos a reunir todas esas funcionalidades en dos ficheros:

- RecibirCan.h: Recogerá todas las funcionalidades requeridas para recibir una trama can en el M68HC12.

- EnviarCan.h: Recogerá todas las funcionalidades requeridas para enviar una trama CAN en el M68HC12.

- El primer paso realizado, consiste en la parametrización y división de todas las funcionalidades en métodos, quedando como sigue:

    - Sincronizar_CAN: Método explicado con anterioridad, en el cual nos matenemos a la espera hasta que el dispositivo se sincroniza con el bus.

    - Establecer_Reloj (velocidad_maquina en MHz, velocidad BUS):
        - En éste método pasamos como parámetros la velocidad del dispositivo en el que se implementa, así como la velocidad del bus, estableciendo en éste el TimeQuanta, TSEG1, TSEG2, PRESCALER, SJW.

    - Comprobar_buffers(): Devuelve para el caso de la emisión, cuál de los tres buffers se encuentra disponible para emitir.

    - Tamanyo_cabecera(): En función de la trama a enviar, calcula el tamaño de ésta para comunicarsela al receptor, evitándole éste trabajo al programador.

    - Tamanyo_Trama(): Al igual que la anterior función, en relación a la trama que se envía en hexadecimal, se calcula su tamaño y se le pasa al receptor.

    - Enviar_trama(cabecera,trama): Una vez la trama lista, utiliza los métodos tamaño_cabecera y tamanyo_trama para calcular sus tamaños, comprueba que buffer está disponible y emite dicha trama.

    - Recibir_trama(*t_cabecera, *t_trama, *cabecera,*trama): Recibe por referencia 4 variables en las cuales se escribirán respectivamente el tamaño de la trama, el de la cabecera, la propia cabecera y la trama.

    - Imprimir_datos(): Imprime utilizando las utilidades seriales, los datos relativos a la última trama recibida.

    - Limpiar_Datos(): Limpiar las variables en las que se almacena la última trama, para asegurarse de que en caso de recibir una trama con menor

cabecera o de menor tamaño, no existan datos residuales que puedan confundir al programador.

- Limpiar_buffers(): Éste método limpia los bufferes que se refieren a las interrupciones, para que una vez habilitados y terminada la recepción se bajan y no se mantenga continuamente el programa tratando de recibir una trama.

3.4 Prueba de las librerías.

Para probar la funcionalidad de ambas librerías, se desarrollan dos ficheros:

3.4.1 Ejemplos simples

- EnviarCan.c: Prueba las funcionalidades del EnviarCan.h.

```
#include "EnviarCan.h"
int main(){
  lock(); // Inicializacion.
    serial_init();
    serial_print( "Iniciando ENVIO POR EL BUS CAN" )
    establecer_reloj(8,8);
 unlock();
 while(1){
        sincronizar_CAN();
        char c;
  c = serial_recv();
        enviar_trama( 0x15555555 , 0x99ULL);
        serial_print( "\n\rTrama enviada" );
 }
}
```

- RecibirCan.C: Prueba las funcionalidades del RecibirCan.h utilizando interrupciones.

```
#include "RecibirCan.h"
int main(){
```

```
  lock();
    serial_init( );
    long TamanoCabecera=0, TamanoTrama=0;
    unsigned char Cabecera[4], Trama[8];
    establecer_reloj(8,8);
       limpiar_buffers();
   limpiar_datos(&TamanoCabecera,&TamanoTrama,Cabecera,trama);
 unlock();
 while(1){
  if(interrupt == 1){
      recibir_trama(&TamanoCabecera,&TamanoTrama,Cabecera,trama);
   imprimir_datos(TamanoCabecera,TamanoTrama,Cabecera,Trama);
   limpiar_datos(&TamanoCabecera,&TamanoTrama,Cabecera,trama);
   serial_print("\n\r");
   interrupt = 0;
   }
 }
}
void __attribute__( ( interrupt ) ) vi_canrx ( void ) {
  interrupt = 1;
  _io_ports[ M6812_CRFLG ] = M6812B_RXF;
}
```

3.4.2 Ejemplo utilizando un display LCD.

- La última prueba realizada, consiste en una pequeña aplicación, la cual se queda continuamente esperando por la emisión de un número de 0 - 100 mediante una interrupción generada por el emisor, relacionado con el porcentaje de combustible existente en un coche.

  Para esto, creo un fichero nuevo llamado Datos.h, en el cual añado los métodos necesarios para utilizar el display, además del métododo convertir_nivel(), en el cual dependiendo del porcentaje recibido se pintará en el display una raya emulando los paneles de los coches actuales.

  - 0-10%: _____. 10-20%: ____*.
  - 20-40%: ___**. 40-60%: __***.
  - 60-80%: _****. 80-100%: *****.

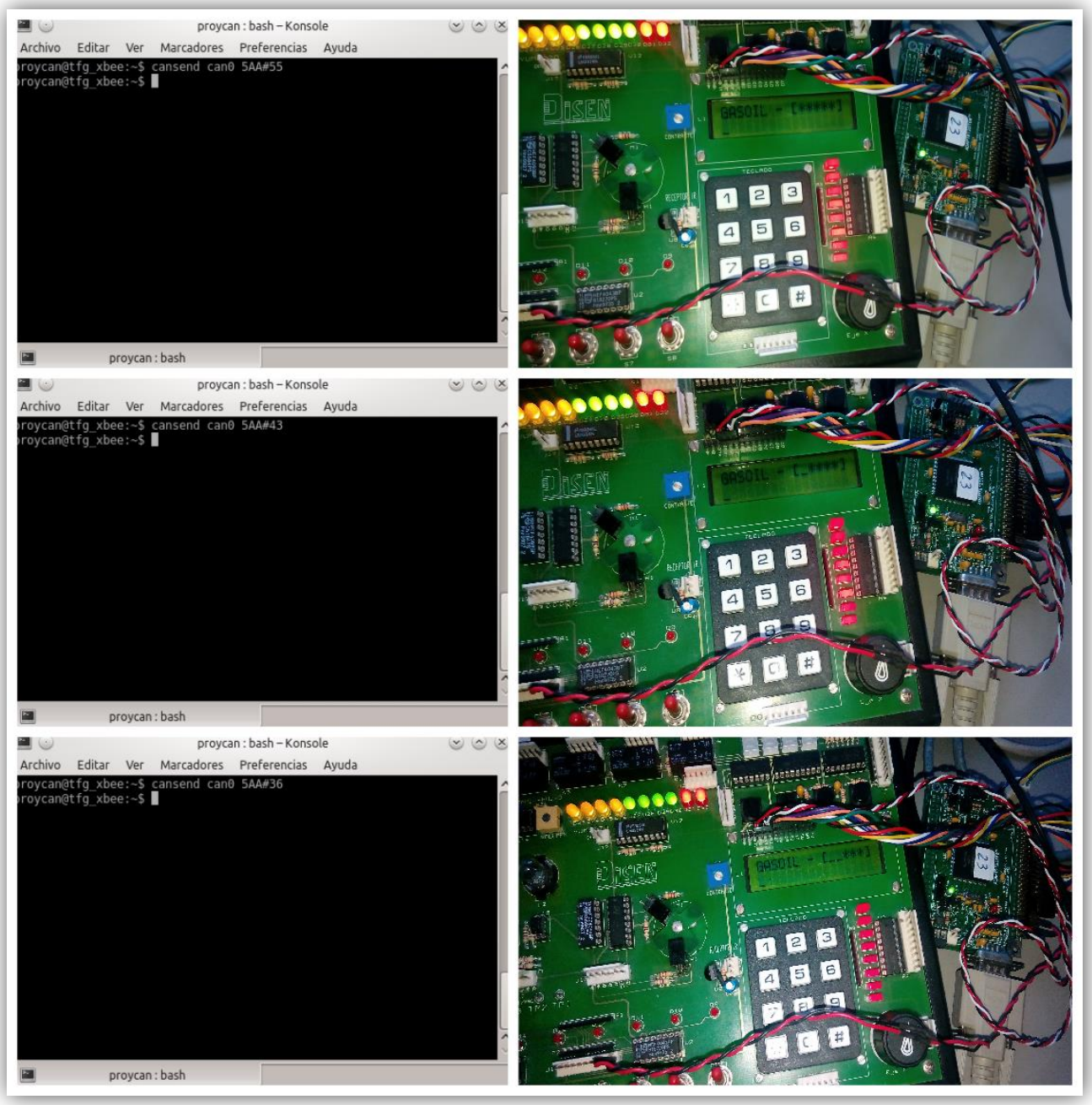


Ilustración 34. Prueba de librerías

Código relacionado del display LCD.

A pesar de que en el repistorio se encuentra un fichero llamado datos.h, en el cual se encuentran todas las funcionalidades con respecto al display LCD, a continuación se muestra el método que pasándole el nivel en hexadecimal extraido de la trama CAN, se utiliza para mostrar en el display el estado del combustible.

```
void convertir_nivel(unsigned char nivel){
    if ( nivel >= 0x50 ) {
     enviaDato ('*');enviaDato ('*');enviaDato ('*');enviaDato ('*');enviaDato ('*');
    }
    else if ( nivel < 0x50 && nivel > 0x3C ){
     enviaDato ('_');enviaDato ('*');enviaDato ('*');enviaDato ('*');enviaDato ('*');
    }
    else if ( nivel < 0x3C && nivel > 0x28 ){
     enviaDato ('_');enviaDato ('_');enviaDato ('*');enviaDato ('*');enviaDato ('*');
    }
    else if ( nivel < 0x28 && nivel > 0x14 ){
     enviaDato ('_');enviaDato ('_');enviaDato ('_');enviaDato ('*');enviaDato ('*');
    }
    else if ( nivel < 0x14 && nivel >0x0A ){
     enviaDato ('_');enviaDato ('_');enviaDato ('_');enviaDato ('_');enviaDato ('*');
    }
    else if ( nivel < 0x0A ){
     enviaDato ('_');enviaDato ('_');enviaDato ('_');enviaDato ('_');enviaDato ('_'); }}
```

Capítulo 4.
Conclusiones y líneas futuras.

4.1 Conclusiones.

Se ha logrado completar el objetivo del proyecto, tanto el primer objetivo, el cual era entender el funcionamiento del bus can, del M68HC12 y del msCAN12, cómo el objetivo final, que consistía en el desarrollo de una librería que reuniera las funcionalidades necesarias para utilizar el bus CAN en el M68HC12.

A pesar de existir una gran cantidad de información del bus CAN y también del microcontrolador M68HC12, no he sido capaz de encontrar ni un solo desarrollo en el que se integren ambas, aunque es cierto que existe un buen manual y alguna que otra página de referencia.

En estos momentos, el potencial que tienen el bus CAN(es utilizado en la gran mayoría de vehículos) y los microcontroladores, los cuales se presentan en casi todos los electrodomésticos que utilizamos a diario, afirman que han sido buenas elecciones para el desarrollo de un proyecto de estas características, ya que te permite ser capaz de llevar a cabo numerosas ideas relacionadas con la materia.

Con respecto al desarrollo y al aprendizaje, he sido capaz de descubrir las infinitas posibilidades que tiene el uso del bus CAN, así como su relativa facilidad de uso. A pesar de que hay aspectos que son un poco difíciles de comprender, cualquier persona con conocimientos básicos de programación y el manual del fabricante sería capaz de realizar algún que otro ejemplo sencillo.

4.2 Líneas futuras.

A pesar de que han sido desarrolladas bastantes funcionalidades, las posibilidades que existen para completar estas funcionalidades son enormes, aunque algunas que se me pasaron por la cabeza durante el desarrollo del proyecto fueron:

- Utilizar filtros de aceptación.
- Uso de interrupciones para el envío.
- Realizar ejemplos con sensores y actuadores utilizando varios microcontroladores que simulen un sistema completo.
- Tratar de utilizar un M68HC12 y el bus can, con un conector OBD para leer lo que ocurre en el bus de un coche.

Capítulo 5.
Conclusions

5.1 Summary.

Once the work is finished, the first conclusion is that although there are a lot of information about the CAN bus and the M68HC12, there isn't information where the CAN bus and the M68HC12 are integrated, although it's true that exists a good manual and some other reference pages.

At present, the potential of the CAN bus and microcontrollers confirm that they have been good choices for the development of a project like this, because it allows to develop many applications related with this area.

About development and learning, I discover all possibilities for using CAN bus, and its ease of use. Although there are aspects that hava a little difficult, but anyone with basic knowledge of programming and the M68HC12 manual could do some simple example.

Capítulo 6.
Presupuesto

A continuación se presentan las tablas de presupuestos relativas al material necesario y al trabajo desarrollado.

Elemento	Número de elementos	Precio unitario	Precio total
Tarjeta CPC_PCI	1	30€	30€
Conector sja1000	2	10€	20€
ADAPT 912 incluyendo M68HC12.	2	150€	300€
Cableado y conectores CAN.	2	10€	20€
Ordenador con Ubuntu	1	200€	200€
-	-	-	-
Investigación	60 horas	35€	2100€
Desarrollo e implementación	20 horas	35€	700€
Cuantía total			**3370€**

Tabla 1. Presupuesto de materiales y mano de obra

Al ser difíciles de encontrar materiales como la tarjeta CPC_PCI, y el conector sja1000, los precios son orientativos basados en productos de similares características.

Bibliografía

[1] Repositorio del código.
https://github.com/ULL-InformaticaIndustrial-Empotrados/TFG_Lorenzo_CAN/blob/master/src/EnviarCan.h

[2] Can4linux
http://www.can-wiki.info/can4linux/man/can4linux_8h_source.html

[3] Configuración bus CAN.
https://developer.ridgerun.com/wiki/index.php/How_to_configure_and_use_CAN_bus

[4] Can utils.
https://discuss.cantact.io/t/using-can-utils/24

[5] CPC_PCI.
http://www.ems-wuensche.com/product/manual/can-pci-plugincard-multiple-channels-cpcpci.pdf

[6] Can4linux.
https://en.wikipedia.org/wiki/Can4linux

[7] BUS_CAN.
https://es.wikipedia.org/wiki/Bus_CAN

[8] BUS CAN (inglés)
https://en.wikipedia.org/wiki/CAN_bus

[9] ADAPT912
http://support.technologicalarts.ca/docs/LegacyProducts/68HC12/Adapt912%20Family/Adapt912B32/912Man1g.pdf

[10] SOCKET CAN
http://ftp.icpdas.com/pub/cd/fieldbus_cd/can/pci/pcm_piso-can_series/driver/linux_can_driver/socketcan/linux_socketcan_can_bus_manual.pdf

[11] BUS CAN
https://www.kernel.org/doc/Documentation/networking/can.txt

[12] BUS CAN
http://elinux.org/CAN_Bus