

Trabajo Fin de Máster Máster Universitario en Ingeniería Aeronáutica

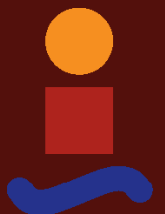
Desarrollo de Plataforma Hardware y Software de una Unidad de Control Electrónico para el Motor del Vehículo FOX

Autor: Pablo Marín Cortés

Tutor: Francisco Gavilán Jiménez

Departamento de Ingeniería Aeroespacial y Mecánica de Fluidos
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018



Trabajo Fin de Máster
Máster Universitario en Ingeniería Aeronáutica

Desarrollo de Plataforma Hardware y Software de una Unidad de Control Electrónico para el Motor del Vehículo FOX

Autor:
Pablo Marín Cortés

Tutor:
Francisco Gavilán Jiménez

Departamento de Ingeniería Aeroespacial y Mecánica de Fluidos
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Máster: Desarrollo de Plataforma Hardware y Software de una Unidad de Control Electrónico para el Motor del Vehículo FOX

Autor: Pablo Marín Cortés
Tutor: Francisco Gavilán Jiménez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Me gustaría agradecer a mis padres, mi hermana y toda mi familia por el apoyo y la comprensión que me han brindado a lo largo de mi periodo de estudios en la Escuela. Asimismo, mi más sincero agradecimiento a todos mis amigos y compañeros de clase, que han estado siempre ahí para sacarme una sonrisa cuando más lo necesitaba.

Esta sección de agradecimientos no estaría de ninguna manera completa sin alabar la dedicación demostrada por Francisco Gavilán en la realización de este proyecto, siendo de inestimable ayuda para llegar a las conclusiones que en él se presentan. Merecen también mi agradecimiento los miembros directivos del FCCL, por la oportunidad brindada para trabajar con el FOX: Carlos Bordons y Miguel Ángel Ridao. Mis agradecimientos también a Juan José Márquez por su tiempo y capacidad para prestar una mano cuando fue necesario.

Resumen

El presente Trabajo Fin de Máster recoge los avances realizados en el proyecto cuyo objetivo es el diseño de una nueva unidad de control electrónica del vehículo FOX, desarrollado en el seno del Departamento de Ingeniería de Sistemas y Automática con la colaboración del Departamento de Ingeniería Aeroespacial y Mecánica de Fluidos, ambos pertenecientes a la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla.

El estudio parte de las conclusiones del Trabajo Fin de Grado "*Estudio Preliminar del Diseño de una Nueva Unidad de Control Electrónico del Motor del Vehículo FOX*" [1]. Estructuralmente, es posible dividirlo en varios bloques: de un lado, la recepción y puesta a punto del equipo adquirido, incluyendo las pruebas funcionales necesarias para la validación de sus características; la integración del bus de datos CAN (*Controller Area Network*) en el computador principal, incluyendo pruebas de comunicación con un microcontrolador externo; por último se propone un montaje experimental que emplea todos los elementos estudiados por separado que constituye el paso inmediatamente anterior a la migración del código de los controladores implantados en el vehículo FOX y a la sustitución de su unidad de control electrónico.

Bajo los capítulos en los que toman forma los anteriores bloques, también se adjuntan anexos que complementan los desarrollos presentados, y que ayudan a reproducirlos si así se requiriera.

Abstract

This Final Master Thesis portrays the goals achieved in the design of a new Electronic Control Unit for the FOX vehicle, developed by the Department of Systems Engineering and Automatization with the collaboration of the Department of Aerospace Engineering and Fluid Dynamics from the "Escuela Técnica Superior de Ingeniería" of the University of Sevilla.

The study starts from the conclusions obtained in the Final Degree Project "*Estudio Preliminar del Diseño de una Nueva Unidad de Control Electrónico del Motor del Vehículo FOX*" [1]. It can be divided in several blocks: on the one hand, the reception and setup of the brand new equipment, including functional tests needed to validate its performance; the integration of a Controller Area Network in the main computer, including communication tests with an external microcontroller; and last, an experimental setup is proposed integrating every aspect studied separately, constituting the immediately previous step before migrating the controller code from the current system installed in the FOX vehicle and the substitution of the ECU.

Under the chapters in which the previous blocks take form, there are also some annexes attached complementing the information presented, which can be used to reproduce the experimental setup if required.

Índice

<i>Resumen</i>	III
<i>Abstract</i>	V
1. Introducción	1
1.1. El proyecto CONFIGURA y el vehículo FOX	1
1.2. Estado inicial del proyecto	2
1.3. Objetivos del TFM	4
2. Recepción del nuevo equipo	5
2.1. PCM-3365	5
2.2. PCM-3910	6
2.3. Montajes experimentales propuestos	7
2.4. Hardware	8
3. Instalación y parcheado del sistema operativo basado en Linux	11
3.1. Pasos previos	12
3.2. Instalación	12
3.2.1. Creación de las particiones del disco	14
3.2.2. Instalación de <i>Slackware</i>	14
3.3. Parcheado y compilación del <i>kernel</i>	16
3.4. Pruebas de rendimiento	18
3.4.1. Paquete <i>rt-tests-1.3</i>	18
3.4.2. Perfil de carga	19
3.4.3. Resultados de las pruebas	19
4. Controller Area Network (CAN)	21
4.1. Nociones sobre la capa física del protocolo CAN	21
4.1.1. MCP2551	21
4.1.2. Niveles lógicos	23
4.1.3. Conectores	24
4.2. <i>PCM-3680I</i>	24
4.2.1. Instalación de <i>drivers</i>	24
4.2.2. Comunicación entre canales	27
4.3. Mbed LPC1768	30
4.3.1. El entorno de desarrollador de Mbed	30
4.3.2. Comunicación entre canales	32
Comunicación serie	33
4.4. Comunicación entre dispositivos	37
4.4.1. Prueba de funcionamiento	37
5. Montaje experimental y código de ejemplo para controlador de motor eléctrico	41
5.1. Procesos e hilos	41
5.2. Montaje experimental del demostrador de concepto	42

5.3.	Programación del controlador	43
5.3.1.	Pthreads (POSIX threads)	43
	Pthread_create	43
	Pthread_join	43
	Pthread_exit	43
	Pthread_cancel	43
5.3.2.	Análisis del servomotor <i>Futaba S3003</i>	45
5.3.3.	Análisis del código del controlador del vehículo	47
	Unidad de Control Electrónico (<i>PCM-3365</i> y <i>PCM-3680I</i>)	47
	Código de la tarjeta de adquisición de datos y actuadora (<i>Mbed LPC1768</i>)	51
5.4.	Caracterización del comportamiento	51
	<i>Cyclictest</i> y <i>Hackbench</i>	51
	Medición del tiempo entre paquetes recibidos	52
6.	Conclusiones y trabajo futuro	59
6.1.	Recapitulación y conclusiones	59
6.2.	Trabajo futuro	59
6.2.1.	Pruebas de funcionamiento	59
Apéndice A.	Prueba de funcionamiento de PCM-3910	61
A.1.	Consideraciones de diseño	61
A.2.	Materiales	62
A.3.	Esquema	62
Apéndice B.	Histogramas y medición de tiempos con MATLAB	63
B.1.	Histogramas	63
	B.1.1. Parámetros necesarios para <i>Cyclictest</i>	63
	B.1.2. Programa MATLAB <i>plot_histograma.m</i>	63
B.2.	Determinación de tiempos entre mensajes CAN	64
Apéndice C.	Códigos del demostrador de concepto	65
C.1.	Unidad de Control Electrónico (<i>PCM3365</i> y <i>PCM-3680I</i>)	65
C.2.	Código de la tarjeta de adquisición de datos y actuadora (<i>Mbed LPC1768</i>)	70
C.3.	Medición del tiempo entre paquetes recibidos	72
	<i>Índice de Figuras</i>	77
	<i>Índice de Tablas</i>	79
	<i>Índice de Códigos</i>	81
	<i>Bibliografía</i>	83

1 Introducción

El presente proyecto es una continuación natural del Trabajo de Fin de Grado *Estudio Preliminar del Diseño de una Nueva Unidad de Control Electrónico del Motor para el Vehículo FOX* [1]. El principal objetivo del mismo fue un estudio de viabilidad para un proyecto de modernización de la unidad de control electrónico (ECU por sus siglas en inglés *Electronic Control Unit*) instalada en el vehículo eléctrico FOX del grupo FCCL (*Fuel Cell Control Lab*) que se encuentra bajo la dirección del departamento de Ingeniería de Sistemas y Automática de la Universidad.

1.1 El proyecto CONFIGURA y el vehículo FOX

De igual manera que en el trabajo presentado en [1], el presente estudio se centra en el vehículo FOX. Este proyecto queda encuadrado dentro de la iniciativa CONFIGURA, promovida por el Centro Nacional del Hidrógeno en conjunto con el Instituto Nacional de Técnica Aeroespacial y la Universidad de Sevilla, que consiste en el control predictivo de microrredes reconfigurables con almacenamiento híbrido y móvil.

Más concretamente, tal y como se puede consultar en [2], el proyecto se centra en el control de energía

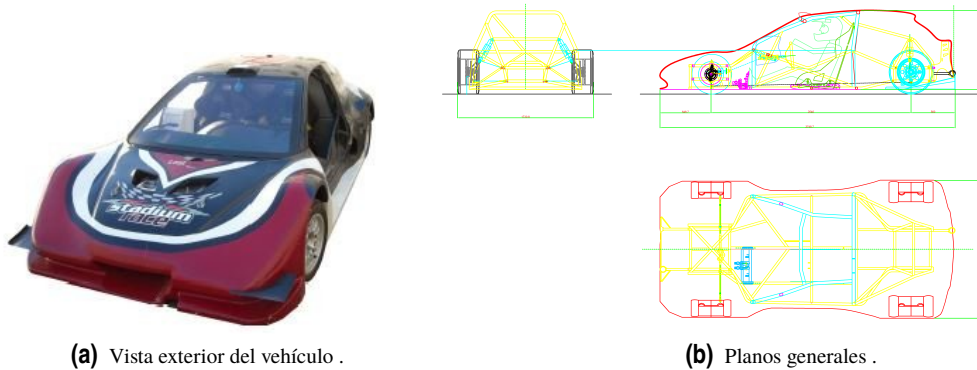


Figura 1.1 Vehículo FOX.



Figura 1.2 Entidades promotoras del proyecto CONFIGURA.

en microrredes (definidas como el conjunto de cargas, generadores y sistemas de almacenamiento que puede ser gestionado de manera coordinada para operar de forma aislada o conectada a la red eléctrica). Se considera especialmente la problemática de la interconexión entre vehículos eléctricos y microrredes, así como el problema de cambio de topología por la conexión/desconexión de cualquier sistema de generación, almacenamiento o carga.

El proyecto pretende desarrollar entre los años 2017 y 2020 distintas estrategias de control en el marco del Control Predictivo para gestionar de manera eficiente la operación de estos sistemas, abordando la reconfigurabilidad tanto en lo referente a los componentes de una microrred, como a microrredes interconectadas. Los sistemas de control también considerarán criterios que incluyan los factores que afectan a la degradación de los sistemas de almacenamiento, con objeto de dotarlos de una mayor durabilidad.

En el ámbito directo de aplicación del presente trabajo, el FOX consiste en un vehículo de competición biplaza adaptado (véase la figura 1.1b) para servir de plataforma de estudio en cuanto a la integración de los vehículos eléctricos y las microrredes anteriormente mencionadas. Las fuentes de energía del mismo pueden ser baterías convencionales o pilas de hidrógeno, clasificándolo como un vehículo eléctrico (EV) híbrido entre un BEV (*Battery Electric Vehicle*) y un FCEV (*Fuel Cell Electric Vehicle*). La planta motriz del FOX consiste en cuatro motores de corriente continua sin escobillas (*brushless*) situados de forma que cada neumático puede ser accionado de manera independiente. La necesidad de un sistema de control automático se justifica por varias razones, entre las que se incluyen lo complicado gobernar la trayectoria del vehículo mediante la actuación directa de cada motor, la gestión de las baterías y la pila de hidrógeno y la seguridad de los ocupantes, así como para la gestión de los datos útiles para la consecución del proyecto.

En cuanto a los elementos de control, el vehículo dispone de volante y pedales compuestos por potenciómetros lineales, así como de una unidad integrada de GPS y medidas inerciales. Toda la información es presentada al conductor a través de una pantalla HMI (*Human Machine Interface*). Es posible encontrar una representación del sistema completo en la figura 1.3.

La unidad de control electrónico del vehículo se encuentra situada tras el asiento del conductor, contenida en un chasis metálico refrigerado que presenta los conectores necesarios para servir de interfaz con todos los elementos del vehículo. Está compuesta por módulos con factor de forma PC/104, un estándar en computadores industriales embarcados, apilados entre sí y conectados con el resto de elementos mediante una red de bus CAN, cuya aplicación resulta un estándar en el mundo automovilístico desde los años 90. El software implementado, como ya se expuso en [1], consiste en la versión 6.3.2 de QNX, un sistema operativo privativo empleado principalmente en sistemas embarcados.

1.2 Estado inicial del proyecto

Varios fueron los objetivos conseguidos al finalizar el proyecto presentado en [1]; por un lado se documentaron aspectos relacionados con el proyecto FOX necesarios para revisar el sistema de la ECU y por otro se llevó a cabo la implementación de un sistema operativo en tiempo real (RT) basado en *Linux* para soportar un entorno exigente en cuanto a tiempos de ejecución, puesto que como se discutió, el control automático preciso y determinista del vehículo depende en gran medida de su robustez. Ello quedó justificado al constatarse que el sistema operativo basado en QNX instalado en el vehículo presenta algunos problemas de corrupción de datos difíciles de predecir que ponen en peligro el desempeño seguro y fiable del controlador, sumado a las características favorables del SO elegido (código libre, extensiva documentación, posibilidad de crear un SO a medida... etc.)

Como se describió en [1, Sección 4.4.1], la dificultad añadida de la idea residió en la necesidad de que el vehículo se encontrara operativo el mayor tiempo posible para la realización de pruebas periódicas en vivo del *software* controlador, lo cual imposibilitó trabajar directamente sobre los elementos que componen la ECU. Con ello, el desarrollo se realizó en un equipo cedido por el departamento de Ingeniería Aeroespacial y Mecánica de Fluidos de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla. Asimismo, este departamento fue el encargado de proporcionar parte de los materiales adicionales necesarios para llevar las pruebas a cabo.

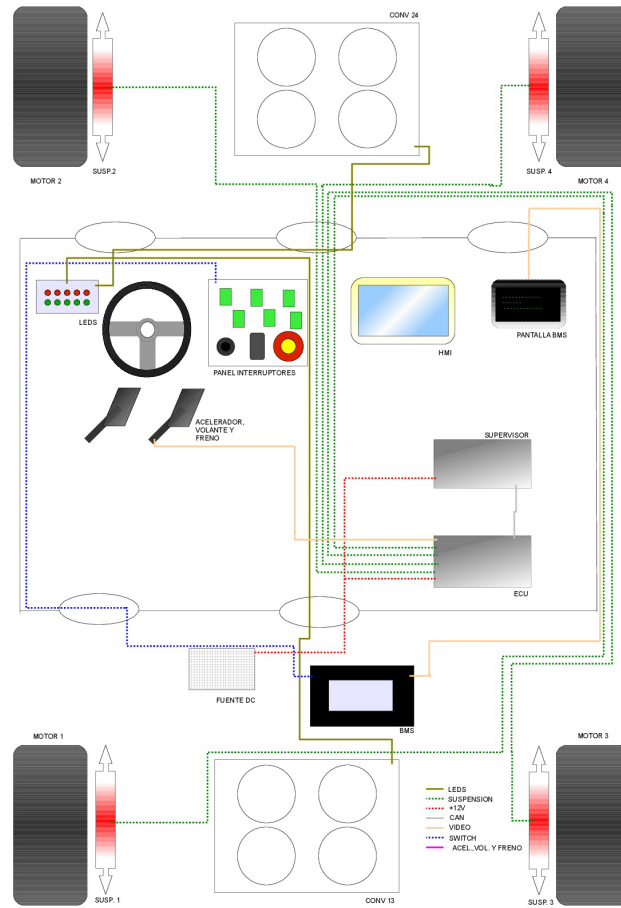


Figura 1.3 Esquema general de las conexiones del vehículo.

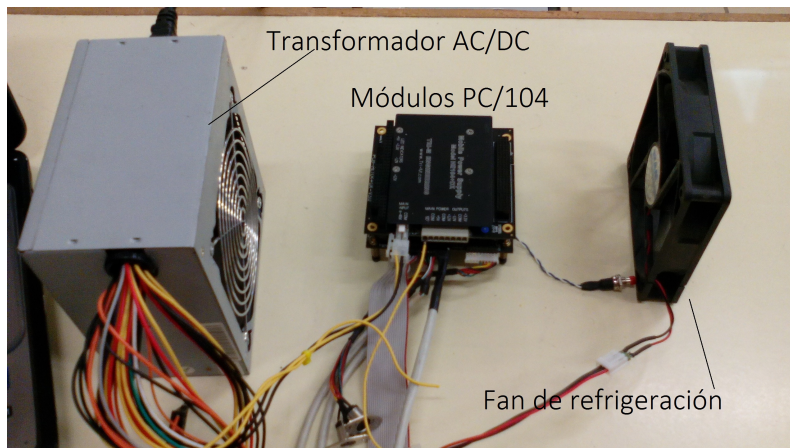
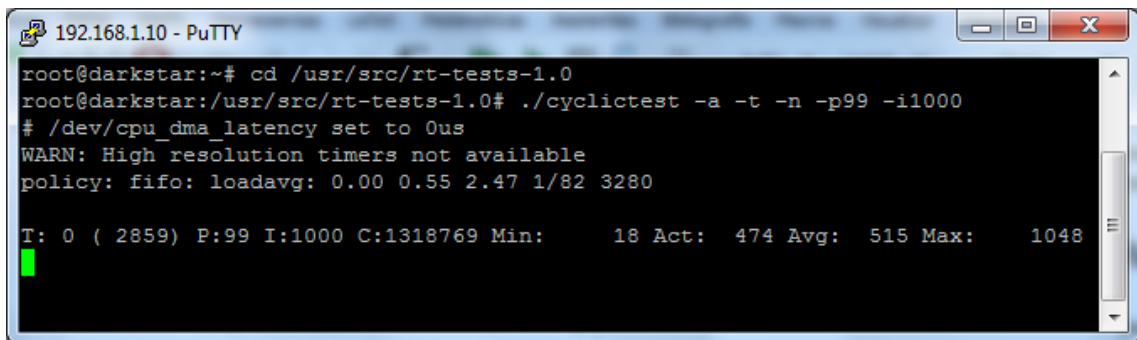


Figura 1.4 Equipo de pruebas empleado en [1].

Los resultados de caracterización del desempeño del sistema que se desarrolló, si bien resultaron prometedoras, pusieron de manifiesto las deficiencias en cuanto al rendimiento de la placa principal de pruebas, lo cual motivó la conclusión de que las capacidades del *hardware* empleado resultaban insuficientes para demostrar el determinismo del sistema. En la figura 1.5 se puede apreciar la pantalla *shell* en la que quedaron reflejados los resultados.

A la vista de las cifras obtenidas, se propusieron una serie de dispositivos que reunían los requisitos



```
192.168.1.10 - PuTTY
root@darkstar:~# cd /usr/src/rt-tests-1.0
root@darkstar:/usr/src/rt-tests-1.0# ./cyclictest -a -t -n -p99 -i1000
# /dev/cpu_dma_latency set to 0us
WARN: High resolution timers not available
policy: fifo: loadavg: 0.00 0.55 2.47 1/82 3280

T: 0 ( 2859) P:99 I:1000 C:1318769 Min:      18 Act:   474 Avg:   515 Max:   1048
```

Figura 1.5 Resultados de latencia.

suficientes como para servir de base a la instalación, llevando a la conclusión de que era posible mejorarlas con un equipo basado en *hardware* actualizado y con mayores prestaciones.

Con todo ello, el principal objeto del proyecto queda marcado por el desarrollo de un sistema de control electrónico determinista que funcione a una frecuencia conjunta de adquisición de datos y procesamiento de la señal que se ha demostrado suficiente para el control del vehículo. Esta frecuencia queda fijada en 50 Hz (tiempo total de 20 ms entre procesamientos sucesivos). Otro de los campos que quedaron fuera del alcance de [1] fue el estudio de la comunicación a través de bus CAN que existe en el vehículo, la cual juega un papel fundamental para conseguir el desempeño deseado.

1.3 Objetivos del TFM

Una vez se ha puesto en contexto el avance realizado en el proyecto de modernización de la ECU, los objetivos concretos marcados para este Trabajo Fin de Máster son entonces los siguientes:

- En primer lugar, la instalación de un SO con capacidades en tiempo real para este nuevo dispositivo, comprobando su desempeño de manera similar a la llevada a cabo en [1, Sección 6.6], que se tratará en el capítulo 2.
- Como se ha comentado, y puesto que el sistema se comunica con los elementos de control del vehículo a través de bus CAN (*Controller Area Network*), el segundo objetivo consiste en añadir esta capacidad de comunicación mediante un módulo PC/104 externo (Capítulo 4) y comprobar que el sistema de comunicaciones resultante demuestra las prestaciones suficientes para la aplicación de control que se está considerando.
- Por último se propone la implementación de un programa de prueba que se ejecute en tiempo real y que, en caso de demostrar que su funcionamiento resulte determinista y estricto en cuanto a tiempos de ejecución, pueda servir de base para la migración de los programas QNX de los controladores del vehículo FOX, y con ello constituir el bloque fundamental de su Unidad de Control Electrónico.

De esta manera comienza el estudio con la descripción del nuevo equipo.

2 Recepción del nuevo equipo

Basándose en las conclusiones expuestas en el Trabajo de Fin de Grado *Estudio preliminar de una nueva unidad de control del motor para el vehículo FOX*, el *Fuel Cell Control Lab* de la Universidad de Sevilla [1] adquiere el siguiente equipo:

- Computador embarcado con factor de forma PC/104 modelo PCM-3365 de la empresa *Advantech*
- Módulo PC/104 de alimentación modelo PCM-3910
- Tarjeta SATA con 64 Gb de capacidad *Transcend MSA370*

Se dispone del siguiente material en el laboratorio:

- Fuente ATX de 350W
- Ventilador de 12V
- Monitor
- USB de 64Gb
- Teclado USB

Además, el laboratorio cuenta con el material electrónico necesario para algunas de las pruebas que se realizarán en capítulos sucesivos.

2.1 PCM-3365

Se trata de un *Single Board Computer* desarrollado por la empresa *Advantech* y lanzado al mercado a mediados del año 2016. Este elemento constituye el verdadero corazón de la unidad de control electrónico del FOX; su misión principal consiste en recibir información relativa al estado de los sensores repartidos por el vehículo, procesarla y enviar la correspondiente señal de control a los motores para conseguir el comportamiento esperado del mismo. Entre otras características que se pueden encontrar en el *datasheet* [3], destacan:

Tabla 2.1 Características de la tarjeta PCM3365.

Procesador	<i>Intel Celeron N2930</i> con velocidad de 1.83 GHz y cuatro núcleos
Memoria RAM	8 GB DDR3L
Conexión Ethernet	RS-232/422/485 desde el conector COM 1, 2 RS-232 desde los conectores COM2/3.
USB	6 puertos (versión 2.0)
Puertos de expansión	Mini PCIe 1 x (Full-size), expansión PC/104 y PCI-104.
Fuente de alimentación	AT/ATX y soporte para alimentación con módulo PC/104.
Tensión de alimentación	5V±5%

A partir de estos datos, junto con los correspondientes al modelo de SBC actualmente implantado en la ECU [1, Sección 2.2.2], es posible predecir en líneas generales que el sistema resultante presentará unas características que superarán a las entregadas por el primero.

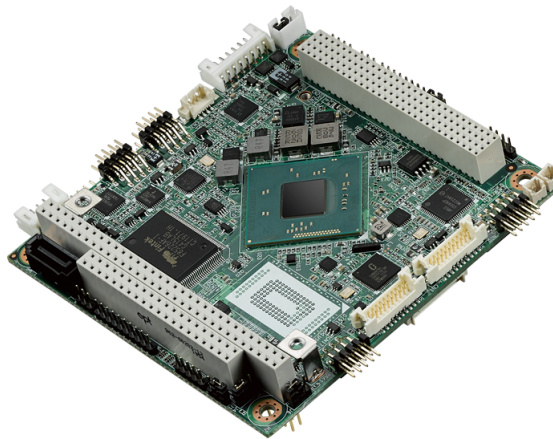


Figura 2.1 SBC PCM-3365.



Figura 2.2 Fuente PCM-3910.

2.2 PCM-3910

Para proporcionar la alimentación necesaria en el vehículo se dispuso asimismo de un módulo con factor de forma PC/104, modelo PCM-3910 también de *Advantech*, que consiste en un convertidor DC/DC con las siguientes características (véase [4]):

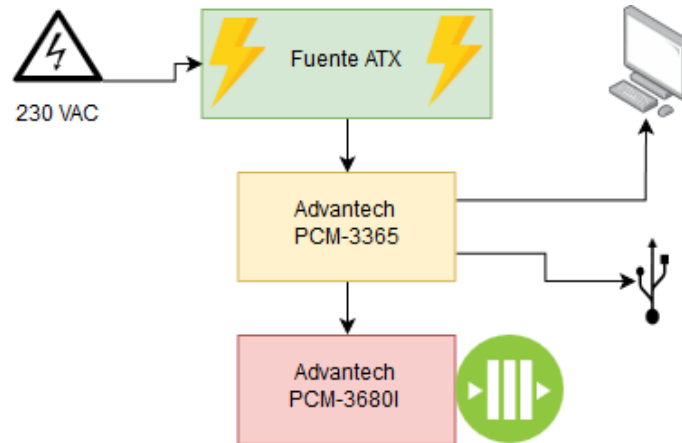
Tabla 2.2 Características de la tarjeta PCM3910.

Potencia nominal	50 W
Rango de tensión de alimentación	10 a 24 VDC \pm 5%
Protección	Polaridad inversa.
Salidas	5 VDC @ 10A
	12 VDC @ 2A
	-5 VDC @ 0.4A
	-12 VDC @ 0.4A

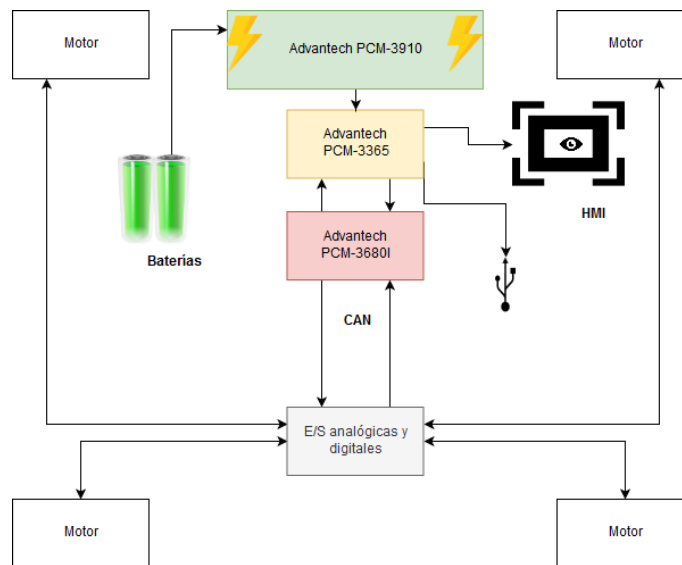
Los datos relativos a los consumos de potencia eléctrica recogidos en la tabla 2.2 son de utilidad para los cálculos presentados en el apéndice A, donde se prueba el funcionamiento de este elemento. En cuanto a los conectores disponibles, conviene notar que dispone de una tornillera de entrada etiquetada como CN1 (*input*) y de puertos de expansión PCI/104 e ISA. La conexión a través de cualquiera de ellos proporciona a la PCM-3365 los voltajes necesarios para su funcionamiento (+5 VDC, +12 VDC y +5 VSB).

2.3 Montajes experimentales propuestos

En este punto conviene describir los dos montajes experimentales que se pretenden llevar a cabo. El primero de ellos consiste en un banco de desarrollo con el objeto de la puesta a punto del equipo, en el que la alimentación del mismo se realiza a través de una fuente ATX convencional, y cuya interfaz gráfica de usuario consiste en una pantalla de ordenador. En el segundo se cambia el módulo de alimentación para adecuarlo a las baterías del vehículo, y la interfaz gráfica es la pantalla integrada del HMI (*Human Machine Interface*). Los diagramas de bloques de ambos se recogen en la figura 2.3



(a) Montaje experimental .



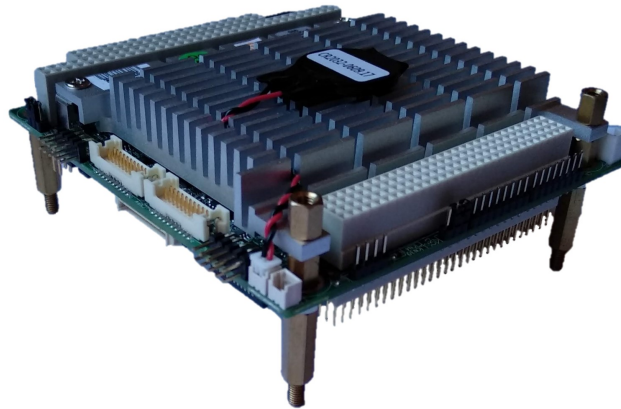
(b) Instalación en el vehículo .

Figura 2.3 Bloques funcionales.

En el caso del montaje en el vehículo, tal y como se discutió en [1], es necesario convertir las señales analógicas procedentes de los pedales y motores en señales digitales que serán transmitidas a la ECU vía bus CAN como entrada para los controladores; de manera inversa, la salida de control en formato digital será convertida a analógica y alimentada a los motores. La conversión analógica/digital y viceversa se realiza mediante los mismos módulos PC/104 de los que dispone actualmente el vehículo: *Advantech PCM-3718HO* para las entradas analógicas y E/S digitales y *Ruby MM-1612* para las salidas analógicas.



(a) Espaciadores incluidos .



(b) Instalados en el equipo .

Figura 2.4 Montaje de los espaciadores hexagonales.

2.4 Hardware

Una vez recogidas las principales características del nuevo material adquirido, se procedió a implantar al nuevo equipo para tener un sistema base sobre el que trabajar. El objeto de esta sección es documentar el proceso llevado a cabo en este sentido.

En primer lugar se instalaron los espaciadores hexagonales que se proporcionan con la placa *PCM-3365* (figura 2.4) con el objeto de separarla de la mesa de trabajo favoreciendo así la ventilación de la misma y evitando cualquier cortocircuito accidental con el espacio de trabajo.

La tarjeta SATA se montó en la ranura situada en la parte inferior del equipo, asegurándola mediante el tornillo incluido al efecto. En la figura 2.5 puede apreciarse con detalle el emplazamiento de los principales componentes de la parte inferior de la placa.

A continuación, y haciendo uso de los manuales del PCM-3365 que se incluyen con el mismo, se identificaron todos los conectores, véase la figura 2.6.

Para el montaje del banco de pruebas se decidió no utilizar el módulo PCM-3910, empleando en su lugar una fuente ATX. Según el manual la alimentación principal del sistema se hace mediante el conector

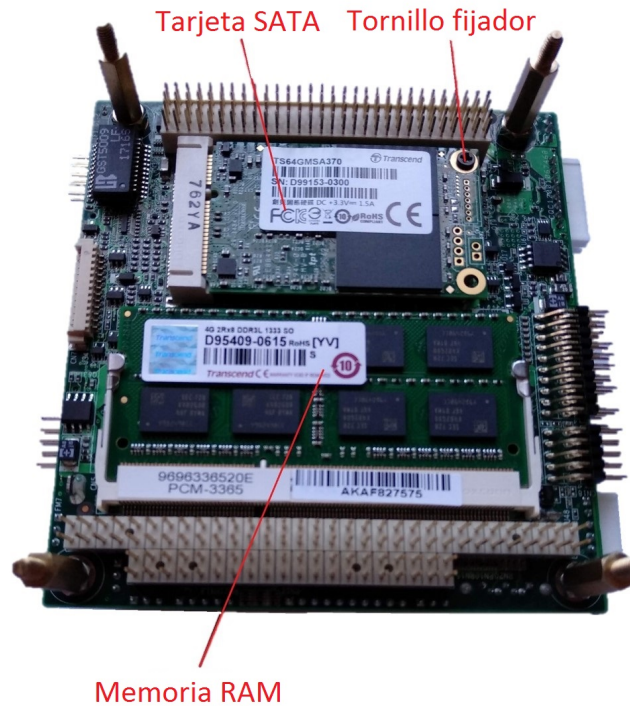
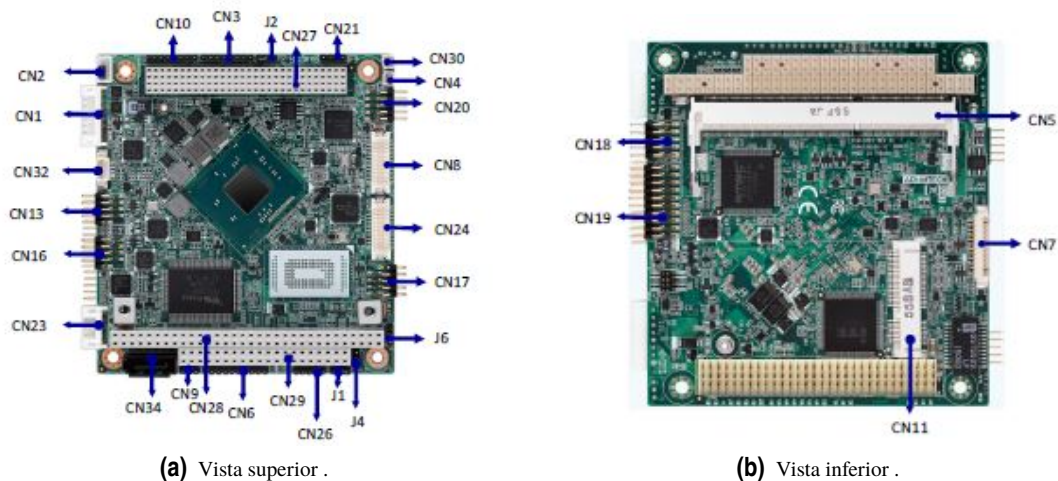


Figura 2.5 Vista inferior de la placa donde se pueden apreciar las ranuras para la tarjeta SATA y para el módulo de memoria RAM.



(a) Vista superior .

(b) Vista inferior .

Figura 2.6 Conectores [3].

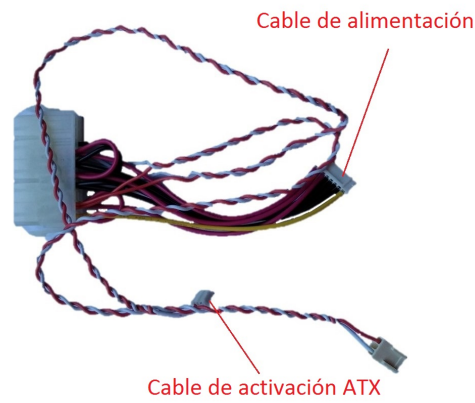
CN1. Además es necesaria la conexión de CN2 (ATX power in) usando el cable de alimentación ATX que viene incluido. En efecto, la consulta del manual (figura 2.7) indica que los pines de este conector son los responsables de mandar la señal de activación a la fuente.

Asimismo se conectó un teclado USB, para lo cual se insertó el adaptador correspondiente en cualquiera de los conectores CN13, CN16 y CN17. Por último se conectó la pantalla en CN7 (conector VGA) (figura 2.8).

CN2	ATX Power In Connector
Part Number	1655303020
Footprint	WHL3V-2M
Description	WAFER BOX 3P 2.0mm 180D(M) DIP 2001-WS-3
Pin	Pin Name
1	+V5SB
2	GND
3	PWR_PSON#

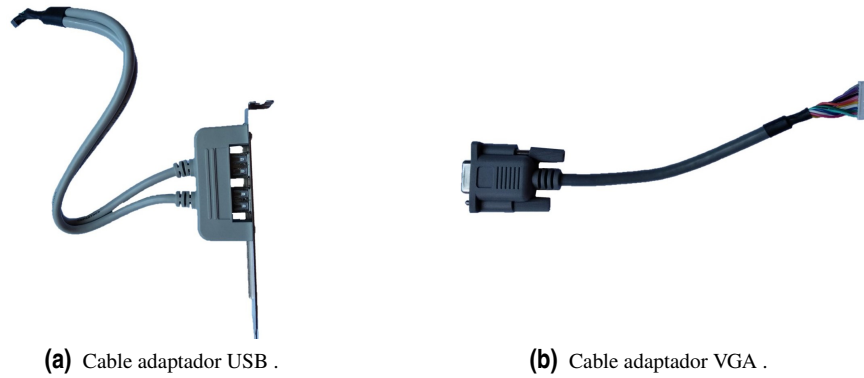


(a) Información sobre CN2 del manual [3].



(b) Cable para ATX .

Figura 2.7 Conexión de la fuente ATX.



(a) Cable adaptador USB .

(b) Cable adaptador VGA .

Figura 2.8 Conexiones restantes .

3 Instalación y parcheado del sistema operativo basado en Linux

Como se estudió detalladamente en [1, Sección 5], un sistema operativo en tiempo real es aquel que garantiza la ejecución de las distintas operaciones que realice dentro de unos márgenes de tiempo lo más deterministas posible, traduciéndose todo ello en una ejecución fiable y predecible. A modo de recordatorio, un sistema operativo en tiempo real se puede clasificar según el grado en el que se aseguren estos tiempos destinados a la ejecución de operaciones críticas: se habla de un sistema operativo de tiempo real estricto cuando se conocen con exactitud, y de tipo *soft* o no estricto cuando importa más la continuidad de la ejecución de las operaciones que el tiempo que éstas disponen para hacerlo. En este sentido, un sistema operativo como *Linux* puede considerarse que no dispone de capacidades en tiempo real estricto. No obstante, existen algunas técnicas de bajo nivel que permiten gestionar las interrupciones del sistema operativo al nivel de su núcleo, permitiendo clasificar el resultante como de tiempo real estricto.

En [1] se decidió emplear el conocido como *parche RT Linux*. Entre otras muchas funcionalidades destinadas a garantizar el comportamiento deseado destaca la inclusión de dos nuevos *schedulers* o programadores de tareas en tiempo real: FIFO (*First In First Out*) y *Round Robin*, que son la base para la programación de *hilos POSIX*, como se estudiará más adelante.

Habiendo introducido el contexto del sistema, la primera tarea en cuanto a *software* consistió en instalar el sistema operativo y aplicar la versión del parche que añade funcionalidades de tiempo real estricto al mismo. Tal y como se estudió en [1, Sección 5.22], se optó por basar la ECU en un SO Linux en su distribución *Slackware*. Se realizó una búsqueda en la página kernel.org para conocer el último lanzamiento del parche RT que coincide con la versión de su *kernel*, que resulta ser la 3.10.17 en el momento en que se escribe este trabajo. Esto restringe la versión más actualizada de *slackware* que es posible instalar en el sistema a la 14.1, lanzada en 2013 si se quiere emplear este tipo de método para obtener un sistema con capacidades RT.



Figura 3.1 Logotipo de *Slackware*.

De esta forma, a modo de resumen, se deben descargar los siguientes archivos:

- Disco de instalación de *Slackware* 14.1
- Código fuente del *Kernel* 3.10.17
- Parche de tiempo real para dicha versión
- Programas de prueba *RT tests*

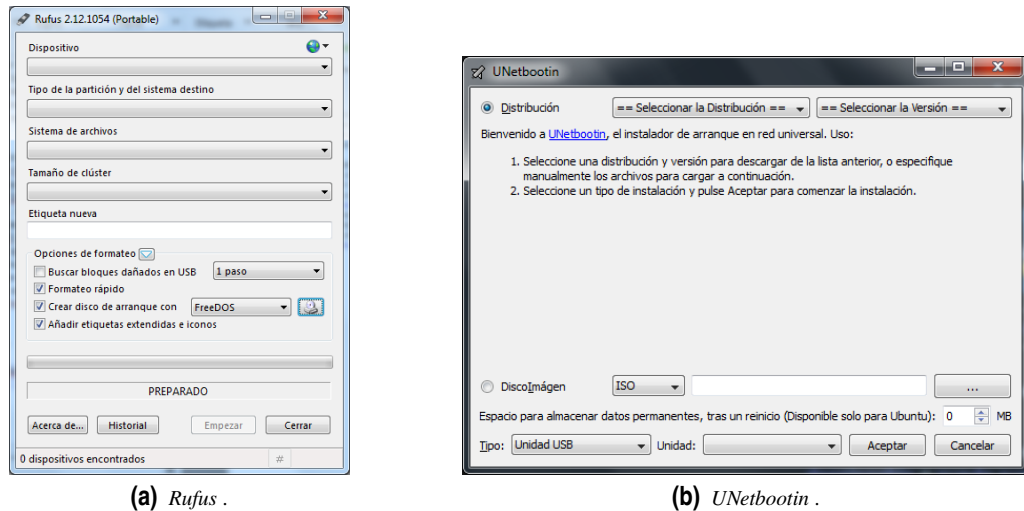


Figura 3.2 Ventanas principales de los dos programas.

- Herramientas de desarrollo
- Librerías de NUMA

Una vez reunidos, puede procederse a instalar la distribución de *GNU/Linux Slackware* en su versión 14.1 en el PCM-3365. El proceso es similar al seguido en [1, Capítulo 6], pero, a diferencia de cómo se procedió en dicho trabajo, se decidió emplear un monitor LCD para visualizar el terminal del dispositivo, simplificando así la manera de trabajar con el equipo.

3.1 Pasos previos

Con el objeto de instalar *Slackware* 14.1 debe arrancarse el sistema desde un medio extraíble formateado adecuadamente. En el presente trabajo se empleó el software *Rufus* [5]. Otro programa útil puede ser *UNetbootin* [6]. A modo ilustrativo se ha decidido explicar el proceso con ambos.

- Se abre el programa *UNetbootin* o *Rufus*. Las ventanas principales de cada uno se presentan en las figuras 3.2a y 3.2b.
- En el caso de *Rufus*, se selecciona el dispositivo extraíble en la primera pestaña y la imagen *ISO* de la distribución seleccionando esta opción junto a "crear disco de arranque con". Por último se hace click en "Empezar".
- Para *UNetbootin* Se selecciona "DiscoImagen" y se busca la ubicación de la imagen *ISO*. Como últimos pasos se selecciona la unidad *USB* donde se encuentra insertado lápiz de memoria y se hace click en "Aceptar"

3.2 Instalación

Para arrancar un computador normalmente se dispone de una serie de botones y LEDs indicadores del estado del sistema en la carcasa donde se encuentra alojada la placa base, componiendo lo que se conoce como panel frontal (*front panel*). En este montaje, sin embargo, no se disponía de este elemento, por lo cual fue necesario conocer la disposición de los pines del conector frontal marcado en el manual como CN10. El esquema se presenta en la figura 3.3

Según este diagrama se puede emular el botón de arranque del sistema puentando momentáneamente los pines 1 (FP_PSIN#) y 2 (GND, tierra) (nótese que el pin número 1 del conector CN10 de la placa está marcado con una pequeña flecha blanca para identificarlo correctamente) . Al hacer esto el equipo manda la

CN10 Front Panel Connector	
Part Number	1653008101-01
Footprint	HD_8x1P_79_D
Description	
Pin	Pin Name
1	FP_PSIN#
2	GND
3	FP_RST#
4	GND
5	Power LED+
6	Power LED-
7	HDD LED+
8	HDD LED-

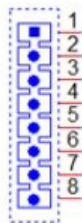


Figura 3.3 Pinout de CN10 [3].

señal de activación a la fuente mediante CN2, que alimenta al sistema a través de CN1.

Tras unos segundos debe mostrarse por pantalla el menú de configuración de la BIOS del PCM-3365 si es la primera vez que se enciende el dispositivo o el menú de arranque de *lilo* si ya se ha instalado la distribución de Linux. Para el montaje de la bancada no se realizaron modificaciones de ningún parámetro por defecto. En este menú es necesario especificar el medio desde el que se pretende realizar el arranque, que en este caso es el USB con el disco de instalación formateado según se ha explicado en la sección anterior.

La primera pantalla que aparece debe ser la de bienvenida del disco de *Slackware*:

```
ISDLINUX 4.06 0x513e7151 ETCD Copyright (C) 1994-2012 H. Peter Anvin et al
Welcome to Slackware version 14.1 (Linux kernel 3.10.17)!

If you need to pass extra parameters to the kernel, enter them at the prompt
below after the name of the kernel to boot (huge.s etc). NOTE: If your machine
is not at least a Pentium-III, you *must* boot and install with the huge.s
kernel, not the hugesmp.s kernel! For older machines, use "huge.s" at the
boot prompt.

In a pinch, you can boot your system from here with a command like:

boot: hugesmp.s root=/dev/sda1 rdinit= ro

In the example above, /dev/sda1 is the / Linux partition.

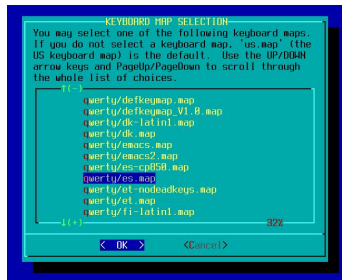
To test your memory with memtest86+, enter memtest on the boot line below.

This prompt is just for entering extra parameters. If you don't need to enter
any parameters, hit ENTER to boot the default kernel "hugesmp.s" or press [F2]
for a listing of more kernel choices. Default kernel will boot in 2 minutes.

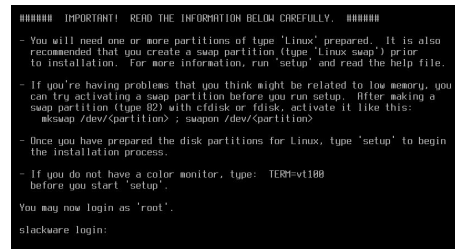
boot: _
```

Figura 3.4 Pantalla de inicio.

Simplemente se presiona *Enter* para ejecutar la versión por defecto del sistema operativo de arranque. A continuación aparece la pantalla de selección de mapa del teclado, que se elige *qwerty/es.map*



(a) Selección de teclado .



(b) Pantalla de inicio de sesión .

Figura 3.5 Ventanas de instalación.

La última pantalla antes de mostrar las opciones principales de instalación permite identificarse en el mencionado sistema operativo de arranque como super usuario (*root*). El siguiente paso es ejecutar el comando *cfdisk*.

3.2.1 Creación de las particiones del disco

El comando *cfdisk* permite crear particiones en el dispositivo de memoria principal en un entorno de menús. Es necesario disponer de dos tipos de particiones del disco duro (tarjeta SATA):

- Una de tipo lógico con sistema de archivos *Linux Swap*. Se utiliza como extensión de memoria RAM cuando ésta se encuentra al límite de su capacidad. El espacio en disco que se ha asignado en este caso es de 2 GB.
- Una primaria con sistema de archivos *Linux ext4*, marcada como *bootable* desde el menú principal. Puede utilizarse la memoria restante tras crear la partición *swap* para crear una sola de tipo primaria o crear hasta cuatro, marcando siempre una de ellas como *bootable*.

Tras crearlas, la lista debería figurar como se indica en la figura 3.6

```

Disk Drive: /dev/sda
Size: 64023257088 bytes, 64.0 GB
Heads: 255 Sectors per Track: 63 Cylinders: 7783
-----
Name      Flags      Part Type  FS Type   [Label]      Size (MB)
-----
sda5          Logical  swap          1998.72*
sda2      Boot      Primary  ext4          62024.52*
-----
[ Bootable ] [ Delete ] [ Help ] [ Maximize ] [ Print ]
[ Quit ] [ Type ] [ Units ] [ Write ]
  
```

Figura 3.6 Particiones necesarias del disco.

3.2.2 Instalación de *Slackware*

Una vez realizadas las particiones se procede a ejecutar el comando *setup* en el *shell*.

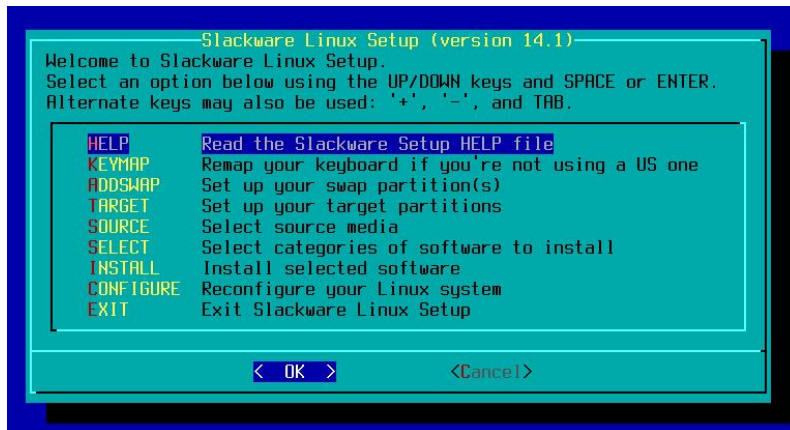


Figura 3.7 Opciones de instalación.

En el apartado TARGET se selecciona la partición en la que se instalará el sistema operativo, que será aquella que se ha creado en *cfdisk* de tipo *Linux*. Se pedirá que se formatee la partición, y se dejará por defecto el sistema de archivos de la misma tras la operación.

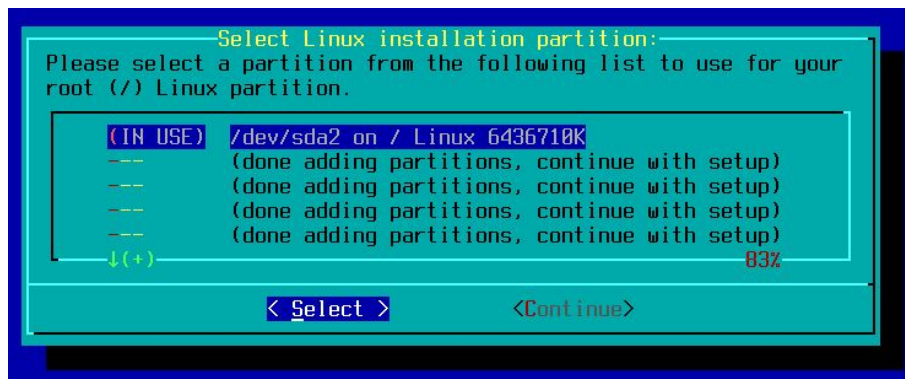


Figura 3.8 Partición objetivo.

Por último se pide que se especifique la ruta en la que se encuentra montado el disco de instalación de Linux, en este caso la del USB:



Figura 3.9 Opción USB.

Se decidió instalar Slackware 14.1 con todos los paquetes incluidos por defecto en la distribución. Tras completar el proceso es necesario reiniciar el dispositivo mediante la orden *reboot*.

3.3 Parcheado y compilación del *kernel*

En esta sección se repite el procedimiento empleado en [1, Sección 6.5], y se puede consultar información adicional en [7]. No obstante se presenta de manera resumida para que sirva como guía para futuras referencias.

Una vez instalado el sistema operativo, el lápiz de memoria USB se formateó en un PC (en el formato de archivos por defecto) para poder transferir los ficheros expuestos en 3 al PCM-3365. Tras conectarlo al puerto USB del sistema (figura 2.8a), hay que montar el dispositivo mediante los siguientes comandos:

Código 3.1 Montar dispositivo USB.

```
root@darkstar:~# mkdir /mnt/usb_device #Se crea el directorio usb_device en la
carpeta /mnt.
root@darkstar:~# mount /dev/sdb1 /mnt/usb_device #Se monta el dispositivo sdb1
en el directorio recién creado.
```

A continuación se creó una carpeta en `/usr/src` para transferir los archivos necesarios para el proceso, empleando el comando `cp`:

Código 3.2 Copia de archivos.

```
root@darkstar:~# mkdir /usr/src/TFM #Nombre de la carpeta creada.
root@darkstar:~# cp /ruta/del/archivo /ruta/de/destino #Se copian los archivos
desde el punto de montaje del USB a la nueva ruta.
```

Dentro de la carpeta mencionada se crearon a su vez una llamada *comp* para guardar el código fuente del kernel y su parche (*linux-3.10.17.tar.gz* y *patch-3.10.17-rt12.patch.bz2*), respectivamente), otra para compilar y almacenar los datos de las pruebas del sistema (*rt-tests-1.3*) y otra para compilar los elementos de la librería NUMA (*utils*).

Se extrajo el kernel en el directorio `/usr/src` y el parche dentro del directorio de las fuentes:

Código 3.3 Extracción del kernel y el parche RT.

```
root@darkstar:~# cd /usr/src/TFM/comp
root@darkstar:/usr/src/TFM/comp# tar -C /usr/src -jxvf linux-3.10.17.tar.gz

root@darkstar:~# cd /usr/src/linux-3.10.17
root@darkstar:/usr/src/linux-3.10.17# bzip2 -d patch-3.10.17-rt12.patch.bz2
```

Se parchearon las fuentes del *kernel* con el comando *patch*. Es importante señalar que el siguiente código debe ser introducido tal y como se presenta en el terminal, puesto que de otra manera el parcheado no se llevará a cabo satisfactoriamente.

Código 3.4 Aplicar el parche.

```
root@darkstar:~# cd /usr/src/linux-3.10.17 # Cambiar al directorio de las
fuentes
root@darkstar:/usr/src/linux-3.10.17# patch -p1 < patch-3.10.17-rt12.patch
```

De esta manera se habrán añadido las funcionalidades de tiempo real explicadas en la sección 6.5 de [1]. Ya se está en condiciones de compilar el *kernel* parcheado mediante los siguientes comandos:

Código 3.5 Proceso de compilación.

```
root@darkstar:~# cd /usr/src/
root@darkstar:/usr/src# rm linux #Elimina el vínculo actual a la carpeta de las
fuentes
```


Código 3.7 lilo.conf.

```

...
image = /boot/vmlinuz-custom-3.10.17 #Nuevo bloque para incluir la imagen del #
      kernel parchado
root = /dev/sda2
# Linux-rt
label = Realtime
read-only

# Linux bootable partition config ends

```

Para aplicar los cambios se escribe *lilo* en el terminal. La opción que se acaba de introducir aparece marcada con un asterisco si el proceso se ha realizado satisfactoriamente.

La última comprobación para saber si se ha compilado correctamente el *kernel* es ejecutar el comando *uname* una vez se arranque el sistema equipado con el mismo.

Código 3.8 Comprobación.

```

root@darkstar:~# uname -r
3.10.17-rt12-realtime

```

3.4 Pruebas de rendimiento

Por último se realizarán los mismos tests de rendimiento empleados para caracterizar el sistema que los empleados en [1, Sección 6.6].

El primero de ellos es *cyclictest*, que se encarga de medir los tiempos de latencia creando procesos y midiendo el tiempo entre interrupciones, y que supone el principal instrumento para conocer el determinismo del sistema. Asimismo, para proveer un entorno de carga exigente se empleó el programa *hackbench*, que se ejecuta en paralelo a la ejecución del código de control y la medición de latencias. Su funcionamiento es simple pero efectivo; envía un número de paquetes en forma de hilos a través del procesador (o procesadores en el caso de tratarse de un sistema de varios núcleos), de un tamaño determinado, siendo ambos parámetros introducidos por el usuario. Un mayor tamaño se traduce en una carga más grande, que junto con el número de paquetes determina el tiempo durante el cual el sistema permanecerá cargado.

3.4.1 Paquete *rt-tests-1.3*

Se trata de la versión más reciente que incluye todas las herramientas necesarias para poner a prueba el sistema. Se pueden conseguir en <https://www.kernel.org/pub/linux/utils/rt-tests/>. Para la compilación es necesario disponer de las librerías *NUMA* (*Non Uniform Memory Access*) y algunos paquetes de desarrollador, que se descargan de www.gnu.org/software/. A continuación se facilita una lista de todas los paquetes y versiones que se han utilizado:

- *numactl* 2.0.12
- *Autoconf* 2.69
- *Automake* 1.16
- *libtool* 2.4
- *m4* 1.4

Todas las instrucciones que hacen referencia a cada paquete se encuentran recogidas en el archivo *INSTALL* correspondiente. Nótese que las versiones empleadas son distintas que en [1, Sección 6.6.1].

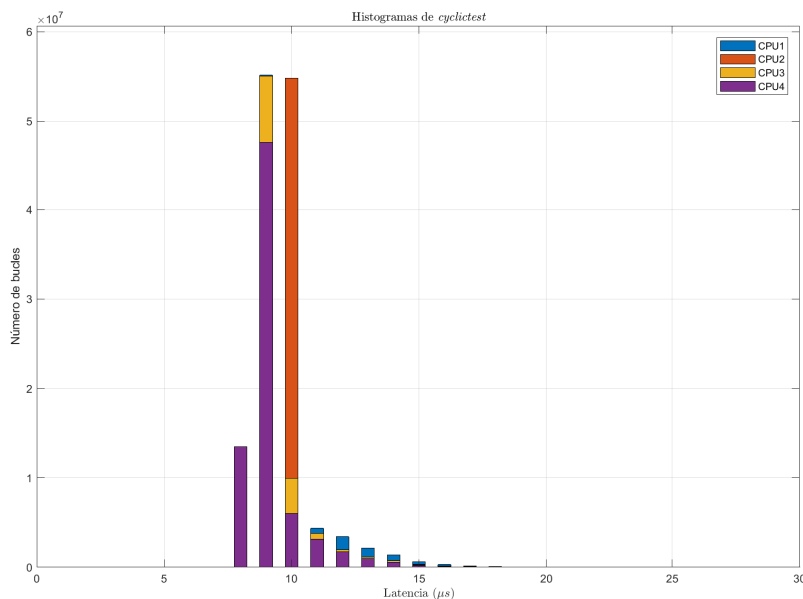


Figura 3.11 Histograma de latencias de los núcleos.

3.4.2 Perfil de carga

Los parámetros y características del ensayo con *hackbench* y *cyclictest* son los siguientes:

- Prioridad de ejecución del hilo de 99%
- Tiempo del ensayo aproximadamente de 24h
- Un hilo por CPU
- Tiempo entre hilos de $1000\mu s$

A diferencia de [1], en el presente trabajo no era posible disponer fácilmente de varias terminales mediante las que ejecutar al mismo tiempo *hackbench* y *cyclictest*. La manera de solucionarlo es ejecutar el primero con los parámetros anteriores y enviarlo al segundo plano. Esto se realiza con la combinación de teclas *CTRL+Z* para detener el proceso y a continuación escribiendo en la terminal *bg* para ponerlo en segundo plano, dejando la terminal libre para poder ejecutar cualquier otro programa al mismo tiempo. Para recuperarlo desde este estado basta con escribir *fg*.

Código 3.9 Ejecución de las pruebas.

```
root@rtlinux:~# cd /usr/src/TFM/rt-tests-1.3
root@rtlinux:/usr/src/TFM/rt-tests-1.3# hackbench -s1000 -l1000000 #Presionamos
CTRL+Z tras ejecutar esta línea.
root@rtlinux:/usr/src/TFM/rt-tests-1.3# bg #Se manda al background
root@rtlinux:/usr/src/TFM/rt-tests-1.3# cyclictest -t -p99 -i1000
```

3.4.3 Resultados de las pruebas

A continuación, en la figura 3.11, se presenta la distribución de latencias conseguida con las condiciones de prueba descritas en las anteriores secciones. Para obtener el gráfico se le indicó a *cyclictest* que escribiera un fichero de texto con todos los datos y éstos se procesaron con un *script* de MATLAB, que puede encontrarse en el anexo B.

El gráfico anterior permite ver una tendencia clara de latencias alrededor de los $10\mu s$. Este resultado es unas diez veces mejor que el obtenido con el equipo original con el que se realizaron las primeras pruebas de caracterización de un sistema en tiempo real (del orden de $500\mu s$, [1, Sección 6.6.3]), poniendo de manifiesto

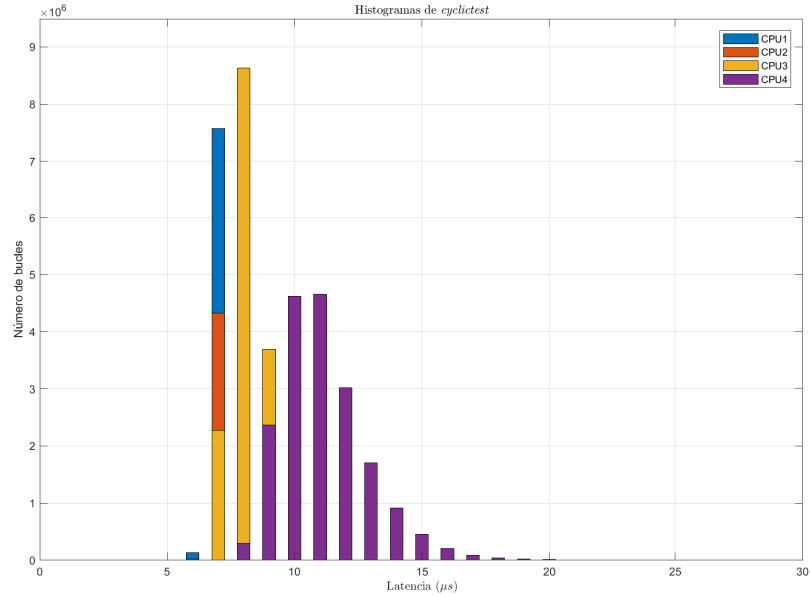


Figura 3.12 Segunda prueba de rendimiento.

la relevancia del uso de un *hardware* y *software* actualizados y con buenas prestaciones en la construcción de un sistema con capacidades en tiempo real. En cuanto al peor escenario, en esta primera prueba se obtuvieron $34 \mu s$, también muy por debajo de la cifra de $1048 \mu s$ obtenida con el primer montaje experimental.

Para validar estos datos se realizó una segunda prueba con los mismos parámetros de carga que en la tabla 3.4.2, pero limitando la latencia máxima representada en el histograma a $60 \mu s$. Los resultados se recogen en la figura 3.12.

Al igual que en el anterior caso, las latencias están distribuidas alrededor de $10 \mu s$, y la más desfavorable se mantiene en el orden, aumentando ligeramente hasta $48 \mu s$. Un comportamiento que se puede observar que el núcleo número 4 presenta un peor rendimiento que el resto, puesto que el número de bucles con latencias mayores es superior al del resto.

Estos resultados resultan más que satisfactorios para implementar una unidad de control electrónico que presente una frecuencia de muestreo típica de 50 Hz (2ms entre muestreos consecutivos), dejando un margen muy amplio para el procesado de la señal de control.

Mediante estas pruebas queda demostrada la idoneidad del sistema adquirido para servir como unidad de control electrónico para el vehículo FOX y para presentar el comportamiento determinista necesario para asegurar un nivel de seguridad suficiente para su correcto funcionamiento, a falta de validar experimentalmente estas afirmaciones. Estos aspectos se estudian en las posteriores secciones.

4 Controller Area Network (CAN)

El siguiente componente de la unidad de control electrónico a estudiar es el bus datos CAN (Controller Area Network). Éste se emplea para hacer posible la comunicación entre la ECU y otros dispositivos que proporcionan el control automático del vehículo, como el supervisor. La manera de implementarlo en el sistema que se ha considerado hasta ahora, formado por la tarjeta principal *PCM-3365*, es mediante una placa controladora adicional compatible. La opción comercial elegida es el módulo con factor de forma *PC/104 PCM-3680I* de *Advantech*, que será estudiada en la sección 4.2.

La verificación del correcto funcionamiento de este sistema de comunicación en la ECU sugirió varias ideas y situaciones. Se planteó la posibilidad de establecer una conexión a través de bus CAN entre la ECU y una tarjeta microcontroladora externa, la *Mbed LPC1768* de *ARM*, estudiada en la sección 4.3, como una manera fiable de confirmar que la comunicación se produce correctamente entre varias capas físicas independientes. Las situaciones consideradas por tanto incluyen la conexión en ambos sentidos de los dos dispositivos considerados y entre sus canales disponibles, en la que se intercambian los papeles de transmisor y receptor.

Los dispositivos disponibles para esta fase fueron:

- Tarjeta CAN *PC/104* de *Advantech*, modelo *PCM-3680I*
- Microcontrolador *Mbed* de *ARM*, modelo *LPC 1768*

Los conocimientos necesarios previos a la descripción más detallada de los dispositivos y sus métodos de interconexión a través de bus CAN son esencialmente los relacionados con la capa física del protocolo, que se describe a continuación en la sección 4.1.

4.1 Nociones sobre la capa física del protocolo CAN

La capa física CAN es aquella que en la representación del modelo OSI (*Open System Interconnection*) se encuentra en el nivel más bajo del protocolo (véase la figura 4.1). Algunas de sus características se encuentran recogidas en el estándar ISO 11898 junto a la capa de datos. Más concretamente, el ISO-11898-2 especifica las subcapas *PMA (Physical Medium Attachment)* y *MDI (Medium Dependent Interface)* de la capa física. Los aspectos relacionados con la señalización física (*PS, Physical Signaling*) se encuentran definidos por la especificación CAN. El diseñador del sistema de comunicaciones puede elegir cualquier transmisor/receptor y medio de transporte de la información mientras que se cumplan los requisitos de *PS*. En este contexto, la Organización Internacional de Estandarización (ISO) ha definido un estándar que incluye la especificación CAN junto con la de la capa física (ISO 11898). Éste fue originalmente creado para las intercomunicaciones de alta velocidad para vehículos a través de CAN. En la figura 4.2 se representa un controlador CAN completo definido por la especificación *dehardware*.

4.1.1 MCP2551

El MCP2551 es un circuito integrado en un paquete *Dual-In-line Package (DIP)* de 8 pines que implementa un transceptor CAN de manera compacta. El *datasheet* de este componente resulta fácil de encontrar online, y es necesario para conocer el *pinout* a la hora de incluirlo en un circuito. La manera de reconocer los pines

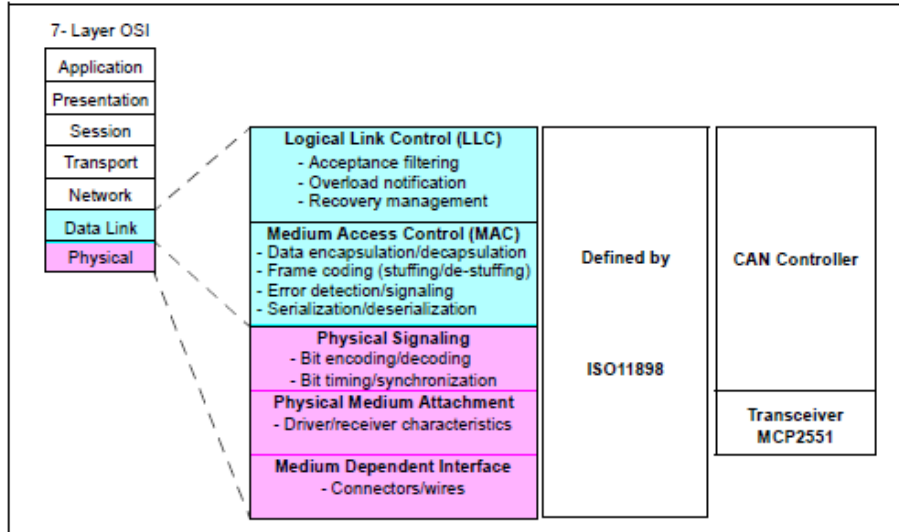


Figura 4.1 Modelo OSI del protocolo CAN [8].

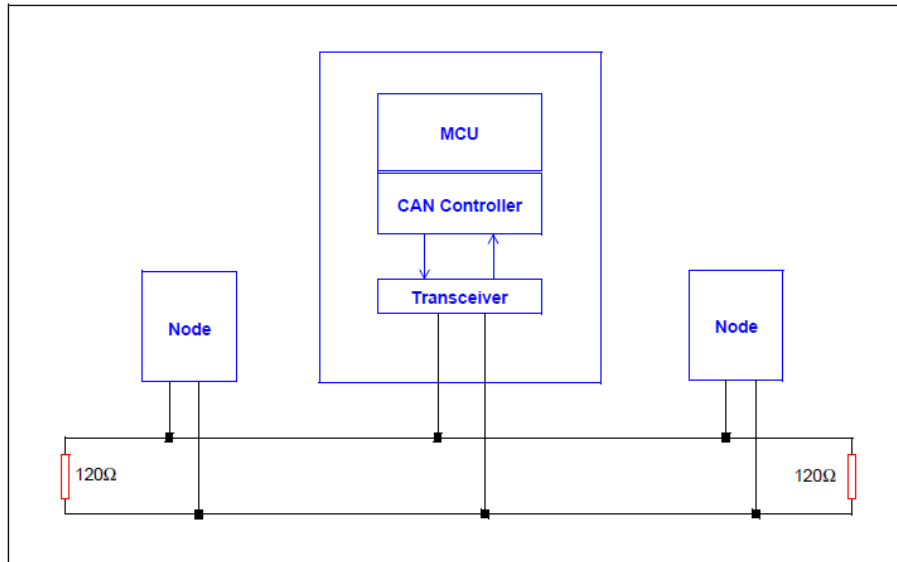


Figura 4.2 Modelo de hardware CAN.

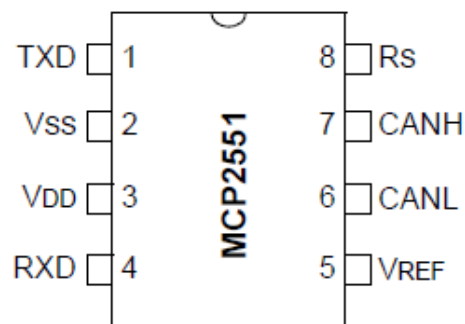


Figura 4.3 pinout del MCP2551.

Tabla 4.1 Tabla de *pinout*.

Pin	Etiqueta	Descripción
1	TXD	Entrada de transmisión de datos
2	V_{SS}	Tierra
3	V_{DD}	Tensión de alimentación
4	RXD	Salida de recepción de datos
5	R_s	Entrada de control de pendiente
6	CANH	E/S de nivel alto CAN
7	CANL	E/S de nivel bajo CAN
8	V_{REF}	Salida de referencia de voltaje

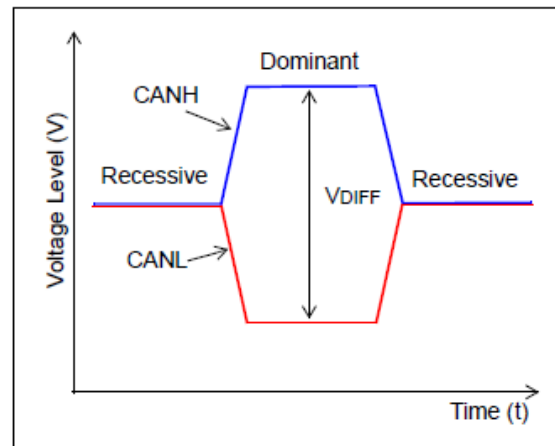


Figura 4.4 Niveles lógicos CAN.

sobre el paquete físico es notar la marca de referencia en la parte superior del mismo. El primer pin a la izquierda de ella recibe el número 1, y los consecuentes pines se suceden en orden ascendente en sentido antihorario, tal y como se muestra en la figura 4.3

En la tabla 4.1 se encuentra la descripción de cada uno de los pines del MCP2551.

Las características principales listadas en el *datasheet* correspondiente [9] se presentan a continuación.

- Soporta velocidades de hasta 1Mbps.
- Cumple con los requerimientos de la normativa ISO-11898 relativos a la capa física del protocolo CAN
- Válido para sistemas de 12V y 24V.
- Pendiente controlada de manera externa para reducir interferencias electromagnéticas (EMI).
- Detección de falta de tierra-

4.1.2 Niveles lógicos

Los niveles lógicos se dividen en recesivo y dominante. El estándar ISO-11898 define un voltaje diferencial para representar estos estados, tal y como se muestra en la figura 4.4

En el estado recesivo (1 lógico en la entrada TXD del MCP2551) el voltaje diferencial entre CANH y CANL es menor que el umbral mínimo ($<0.5V$ en la entrada del receptor o $<1.5V$ en la salida del transmisor, tal y como se aprecia en la figura 4.5); en el estado dominante (0 lógico) el voltaje diferencial en CANH y CANL es mayor que el umbral mínimo. Un bit dominante sobrescribe a un bit recesivo en el bus para conseguir un arbitraje no destructivo mediante bits. Como puede observarse en la anterior figura, cuando el voltaje diferencial de la salida se encuentra entre $1.5V$ y $3V$ se considera un nivel lógico dominante, mientras que el estado será recesivo cuando se encuentre entre $-0.5V$ y $0.05V$. Del mismo modo, en el caso de la entrada se aprecia que el rango se extiende desde los $0.9V$ a los $5V$ para el 1 lógico y entre los $-1V$ y $0.5V$ para el 0 lógico.

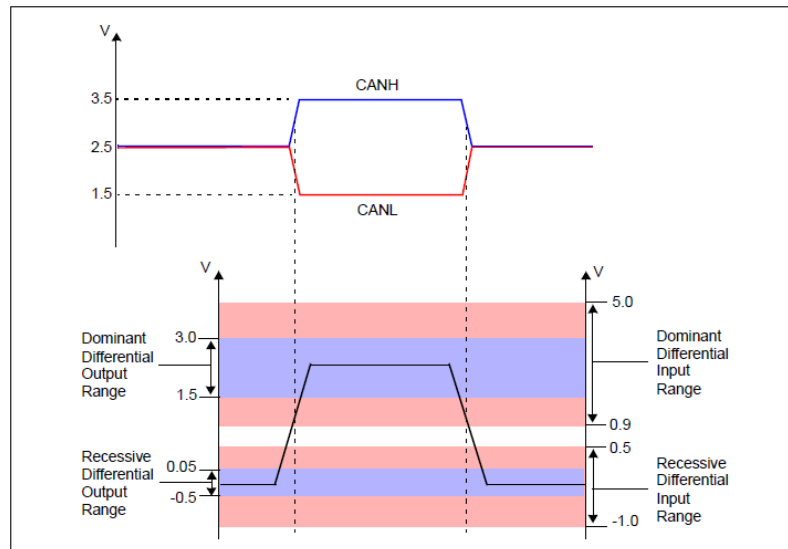


Figura 4.5 Umbrales para el voltaje diferencial de transmisor y receptor en estados recesivo y dominante.

Tabla 4.2 Características de la tarjeta *PCM-3680I* .

Velocidad de transmisión	Hasta 1 Mbps
Frecuencia del controlador	16 MHz
Protección	Aislamiento óptico
Leds indicadores	Uno por puerto
Temperatura	Amplio rango de operación
Sistemas operativos soportados	WinXP/Vista/7 y Linux (32 y 64 bits)
WinCE	5.0/6.0

4.1.3 Conectores

Aunque el estándar ISO-11898-2 no especifica qué tipo de conectores han de emplearse, sí se indica que aquellos que se empleen deben cumplir con los requisitos eléctricos de la especificación. Asimismo, se requieren resistencias de terminación de un valor nominal de 120Ω en cada extremo del bus con el objeto de evitar "rebotes" de la señal como consecuencia de un desajuste de impedancias.

4.2 *PCM-3680I*

Este módulo es el encargado de añadir físicamente la capacidad de comunicación mediante bus CAN al sistema. Se trata de una tarjeta con factor de forma PC/104 con dos canales de bus de datos CAN independientes. Sus principales características son las reflejadas en la tabla 4.2 [10].

Asimismo, este módulo dispone de un puerto PCI/104 para conectar a la computadora PCM-3365.

4.2.1 Instalación de *drivers*

Con el objeto de gestionar la red de comunicación es necesario instalar los controladores o *drivers* del dispositivo *PCM-3680I*. Junto a la misma se incluye un disco de instalación, pero tras consultar su documentación se llegó a la conclusión de que los *drivers* solo son válidos para versiones del *kernel* 2.6.x.x. Por tanto, fue necesario acudir al sitio web de *Advantech* para encontrar los controladores más recientes. Respecto a las versiones soportadas, se encontró la información recogida en la tabla de la figura 4.7; como es posible observar, las pruebas realizadas por los desarrolladores de estos controladores emplearon versiones del *Kernel* similares a la que se implementó en el anterior capítulo (recuérdese que era la 3.10.17). Además, en el apartado relativo a soporte se indica que son válidos para versiones 2.6.33 y posteriores, lo cual concuerda con la instalada.

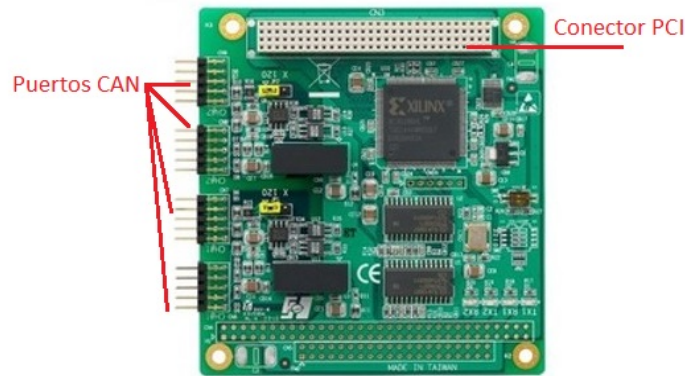


Figura 4.6 Módulo CAN PCM-3680.

2. Test environment

Distribution	kernel version
Ubuntu 12.04	3.2
Ubuntu 14.04	3.13
Ubuntu 15.10	4.2
openSUSE Leap42.1	4.1.12
Fedora 23	4.2.3
Debian 8.3	3.16.0

warning : fedora20 and redhat7.2 default not enable SocketCAN

Figura 4.7 Soporte de las versiones del Kernel.

En <https://www.advantech.com/products> se encuentra el repositorio con los *drivers* más recientes. Los archivos de los que es necesario disponer son un archivo comprimido *CAN cards (PCI-1680_MIC-3680_PCM-3680I)_advSocketCAN_V1.0.0.0.tar.gz* y el documento de texto con la información relativa a la instalación. Cabe comentar que para facilitar el manejo del archivo en el sistema de archivos de Linux es posible autocompletar el nombre del mismo escribiendo los primeros caracteres y pulsando la tecla *TAB*.

Código 4.1 Extracción de *drivers*.

```
root@rtlinux:~# cd /usr/src/TFM
root@rtlinux:/usr/src/TFM# tar -zxvf CAN cards (PCI-1680_MIC-3680_I_)\textit{PCM-3680I}\_advSocketCAN\_V1.0.0.0.tar.gz
root@rtlinux:/usr/src/TFM# cd advSocketCAN_V1.0.0.0/\textit{drivers}
root@rtlinux:/usr/src/TFM# make #Se compilarán los archivos advsocketcan.ko y advcan_sja1000.ko
root@rtlinux:/usr/src/TFM# make install #Con este comando se instalan los \textit{drivers}
```

El siguiente aspecto a considerar fue que estos *drivers* se cargaran automáticamente durante el arranque del equipo. Para ello es necesario modificar el fichero encargado de hacerlo, esto es, *rc.modules-3.10.17-smp* del directorio */etc/rc.d* empleando cualquier editor de texto de *Linux*. Para más información sobre el uso de *vim (Vi IMproved)* se recomienda la consulta de [1]. Sin embargo, la versión 14.1 de *Slackware* incluye *nano*, que posee más funcionalidades, por lo que se empleará a lo largo de todo el presente proyecto.

```

echo "Updating module dependencies for Linux $RELEASE:"
/sbin/depmod -a
else
echo "Module dependencies up to date (no new kernel modules found)."
fi
else # we don't have find, or there is no existing modules.dep, or it is out
date.
echo "Updating module dependencies for Linux $RELEASE:"
/sbin/depmod -a
fi
### Can driver:
# Drivers de la tarjeta CAN de ADVANTECH
/sbin/modprobe lp
/sbin/modprobe can
/sbin/modprobe can_dev
/sbin/modprobe can_raw
/sbin/modprobe advcan_sja1000
/sbin/modprobe advsocketcan

### Mouse support:
# PS/2 mouse support:
# The default in Slackware is to use proto=imps because that works with the
# most types of mice out of the box. For example, using proto=any will

```

Figura 4.8 Carga de *drivers*.**Código 4.2** Carga automática de *drivers*.

```

root@rtlinux:~# cd /etc/rc.d
root@rtlinux:/etc/rc.d# nano rc.modules-3.10.17-smp

```

Los nombres de los módulos adicionales a cargar se encuentran descritos en el archivo de texto que se proporciona con los *drivers*. El aspecto del fichero tras la modificación debería ser el que aparece en la figura 4.8 (téngase en cuenta que sólo se muestra el fragmento modificado). Para futuras referencias se transcribe el contenido del fichero considerado en el código 4.3, que como puede observarse en la figura 4.8 se decidió insertar justo al principio de todos los módulos ya incluidos:

Código 4.3 Inclusión de *drivers* CAN.

```

### Can driver:
# \textit{drivers} de la tarjeta CAN de ADVANTECH
/sbin/modprobe lp
/sbin/modprobe can
/sbin/modprobe can_dev
/sbin/modprobe can_raw
/sbin/modprobe advcan_sja1000
/sbin/modprobe advsocketcan

```

No hay que olvidar reiniciar el equipo tras la modificación de estos archivos para que los *scripts* se ejecuten. La primera manera de comprobar que la operación ha sido llevada a cabo con éxito es que durante la carga del sistema operativo aparecerá momentáneamente un gráfico compuesto por caracteres ASCII en el que se da la bienvenida a los *drivers* de la tarjeta CAN y se comprueban ambos canales disponibles. La segunda consiste en ejecutar el comando *ip link list* en el terminal. Esta orden (*ip*) junto con el sufijo (*link*) muestra las interfaces de la capa de enlace de datos del sistema, esto es, aquella que recibe las peticiones de la capa de red y hace uso de los servicios de la capa física por donde se transmite la información. Con los nuevos *drivers* incluidos deben aparecer dos nuevas interfaces, *can0* y *can1*, correspondientes a los dos canales CAN de los que se dispone. Además, por defecto estas interfaces estarán desactivadas. Todos estos aspectos

```

root@rtlinux:/etc/rc.d# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN mode D
EFAULT qlen 1000
    link/ether 00:0b:ab:df:2d:86 brd ff:ff:ff:ff:ff:ff
3: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT qlen 10
    link/can
4: can1: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT qlen 10
    link/can

```

Figura 4.9 Lista de enlaces de datos.

```

root@rtlinux:~# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN mode D
EFAULT qlen 1000
    link/ether 00:0b:ab:df:2d:86 brd ff:ff:ff:ff:ff:ff
3: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UNKNOWN mode DEF
AULT qlen 10
    link/can
4: can1: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UNKNOWN mode DEF
AULT qlen 10
    link/can

```

Figura 4.10 CAN activado.

aparecen en el terminal tal y como se indica en la figura 4.9, donde se puede comprobar el estado DOWN en la línea de cada interfaz CAN.

Este es el estado en el que ambos enlaces se encontrarían cada vez que se arranque el sistema. Con el objeto de evitar esto se realizó un *script* que se ejecuta en el arranque y que los activa con los parámetros recomendados en la documentación de los *drivers*. La manera de llevar a cabo esto es añadir las siguientes líneas al archivo *rc.local*:

Código 4.4 Modificación en el fichero rc.local.

```

ip link set can0 up type can bitrate 125000
ip link set can1 up type can bitrate 125000

```

Con ello lo que se está haciendo es activar cada una de las interfaces (can0 y can1) y asignándoles una velocidad de 125000 bits por segundo.

Ejecutando de nuevo `ip link list` se obtiene por pantalla el contenido de la figura 4.10.

Con esto finaliza el proceso de instalación de los *drivers* de la tarjeta PCM-3680I. Una de las formas más sencillas posibles de comprobar la interfaz CAN consiste en intercomunicar los dos canales disponibles.

4.2.2 Comunicación entre canales

El árbol o *tree* de directorios del código fuente de los *drivers* contiene una carpeta de ejemplos (*advSocketCAN_V1.0.0.0/examples*) donde se encuentra un fichero mediante el que es posible probar el desempeño de las comunicaciones CAN de la tarjeta PCM-3680I. En el apartado 5 de la documentación incluida con los *drivers*, véase [11], se encuentra la línea de comandos necesaria para compilar los *scripts* de lectura y escritura de mensajes en el bus, que simplemente consiste en emplear el comando `make` en el directorio *advSocketCAN_V1.0.0.0/examples*.

Para intercomunicar los canales basta con conectar sus terminales mediante los cables RS-232 DB9 macho y hembra incluidos con la tarjeta. Para que no exista interferencia entre los elementos de fijación, es necesario eliminar las tuercas hexagonales de la carcasa de uno de ellos.

En primer lugar se ejecuta el programa de lectura (*readCAN*) y se deja ejecutando en segundo plano con los comandos habituales del sistema de archivos de Linux:

Código 4.5 Programa readCAN.

```
root@rtllinux:/usr/src/TFM/advSocketCAN_V1.0.0.0/examples# ./readCAN can0
#Se presiona la combinación ctrl+z para detener el proceso.
root@rtllinux:/usr/src/TFM/advSocketCAN_V1.0.0.0/examples# ./readCAN bg #Se envía
a el proceso al fondo
```

Nótese como la manera de correr el programa consiste en incluir su nombre precedido de *./*, y que en este caso se ha elegido el canal *can0* para la lectura de mensajes. Ahora se ejecuta el programa de escritura *sendCAN* en el otro canal (*can1*). A continuación se adjunta su código para analizar el mensaje que se escribe en el bus.

Código 4.6 Programa readCAN.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5
6  #include <net/if.h>
7  #include <sys/types.h>
8  #include <sys/socket.h>
9  #include <sys/ioctl.h>
10
11 #include <linux/can.h>
12 #include <linux/can/raw.h>
13
14
15 int main(int argc, char **argv)
16 {
17     int s;
18     int nbytes;
19     struct sockaddr_can addr;
20     struct can_frame frame;
21     struct ifreq ifr;
22
23     if (argc != 2)
24     {
25         printf("usage : sendCAN <CAN interface>\n");
26         printf("ex. sendCAN can1\n");
27         return 0;
28     }
29
30     char *ifname = argv[1];
31
32     if((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0)
33     {
34         perror("Error while opening socket");
35         return -1;
36     }
```

```

root@rtlinux:/usr/src/TFM/advSocketCAN_V1.0.0.0/examples# ./readCAN canBAC
using can0 to read
^Z
[1]: Stopped ./readCAN can0
root@rtlinux:/usr/src/TFM/advSocketCAN_V1.0.0.0/examples# by
[1]: ./readCAN can0 &
root@rtlinux:/usr/src/TFM/advSocketCAN_V1.0.0.0/examples# ./sendCAN canI
using can1 to write
root@rtlinux:/usr/src/TFM/advSocketCAN_V1.0.0.0/examples#
ID=0x123 DLC=8
data[0]=0x1
data[1]=0x2
data[2]=0x3
data[3]=0x4
data[4]=0x5
data[5]=0x6
data[6]=0x7
data[7]=0x8
^C
root@rtlinux:/usr/src/TFM/advSocketCAN_V1.0.0.0/examples#

```

Figura 4.11 Resultado de la comunicación CAN.

```

37
38 strcpy(ifr.ifr_name, ifname);
39 ioctl(s, SIOCGIFINDEX, &ifr);
40
41 addr.can_family = AF_CAN;
42 addr.can_ifindex = ifr.ifr_ifindex;
43
44
45 if(bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0)
46 {
47 perror("Error in socket bind");
48 return -2;
49 }
50
51 frame.can_id = 0x123;
52 frame.can_dlc = 8;
53 frame.data[0] = 0x1;
54 frame.data[1] = 0x2;
55 frame.data[2] = 0x3;
56 frame.data[3] = 0x4;
57 frame.data[4] = 0x5;
58 frame.data[5] = 0x6;
59 frame.data[6] = 0x7;
60 frame.data[7] = 0x8;
61
62 printf("using %s to write\n", ifname);
63 nbytes = write(s, &frame, sizeof(struct can_frame));
64
65 return 0;

```

A partir de la línea 47 se pueden ver los valores asignados a las distintas variables fundamentales del marco CAN: identificador, tamaño del mensaje y bits de datos, que son los números hexadecimales desde el 0x1 hasta el 0x8. El análisis del código de *readCAN* revela que estos bits de datos aparecerán por pantalla en cuanto sean recibidos por el canal de lectura.

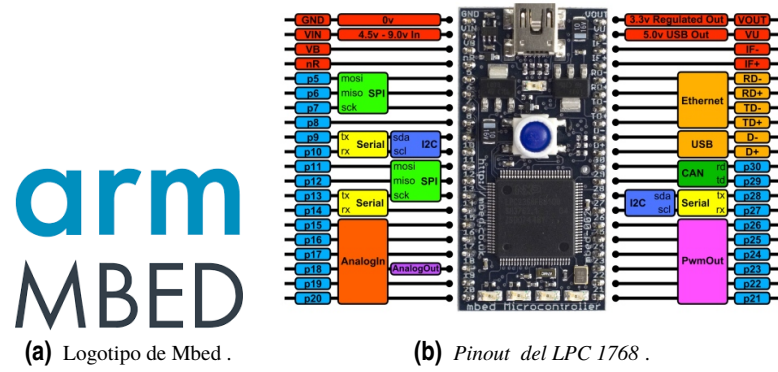


Figura 4.12 Detalles del microcontrolador.

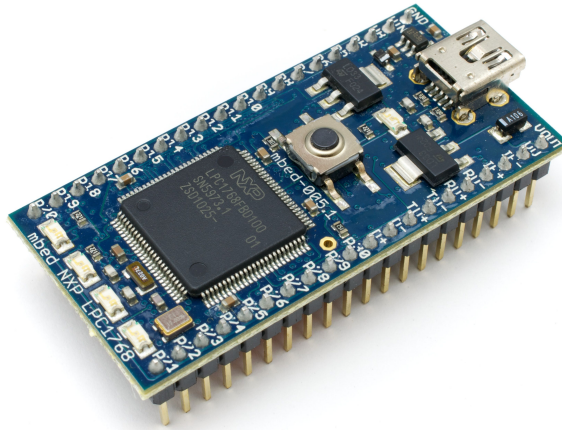


Figura 4.13 Apariencia del dispositivo.

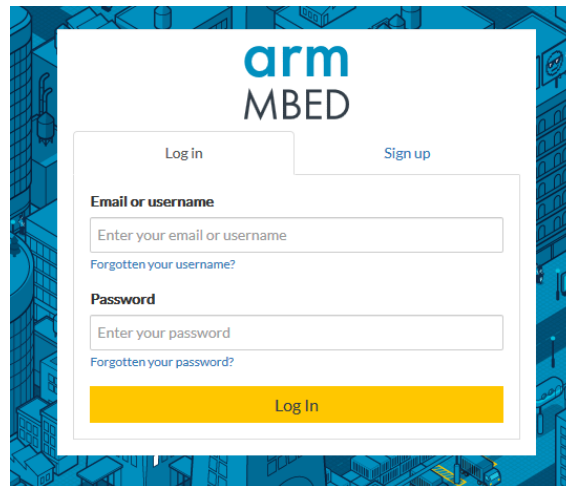
4.3 Mbed LPC1768

Una vez comprobada la capacidad del sistema de comunicarse a través de bus CAN, y con el objetivo de ensamblar un entorno de prueba más realista del nuevo sistema, se consideró la posibilidad de que el conjunto PCM-3365/PCM-3680I se comunicara con un dispositivo externo. Para ello se empleó el *Mbed LPC1768* de ARM, un microcontrolador con pequeño factor de forma pero con potentes prestaciones, entre las cuales se incluyen dos canales de bus CAN. La particularidades de esta plataforma son que es sencilla de programar y que posee un compilador online que facilita la portabilidad de los códigos creados. También posee una extensa comunidad de soporte, lo cual la hace idónea para la implementación rápida de un banco de pruebas.

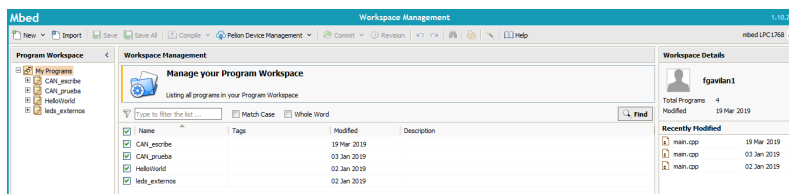
Para que sirva de referencia a lo largo de la descripción del montaje experimental que se llevó a cabo, se presenta a continuación en la figura 4.12b el *pinout* de la placa, que se puede encontrar en una tarjeta incluida con la misma o en [12]. Es necesario aclarar que este dispositivo posee dos canales CAN independientes, aunque en la figura 4.12b solamente aparezca uno correspondiente a los pines 29 y 30. En efecto, como se recoge en la siguiente sección, el otro canal se establece entre los pines 9 y 10.

4.3.1 El entorno de desarrollador de Mbed

Como introducción al dispositivo, se describe brevemente el entorno de desarrollo integrado (*IDE*) de *Mbed* y la manera de compilar y cargar un *script* en el mismo. Como ya se ha comentado, es completamente *online*, por lo que no es necesario instalar ningún *software* en el equipo desde el cual se accede.



(a) Pantalla de registro .



(b) Ventana principal del compilador .

Figura 4.14 Ventanas de instalación.

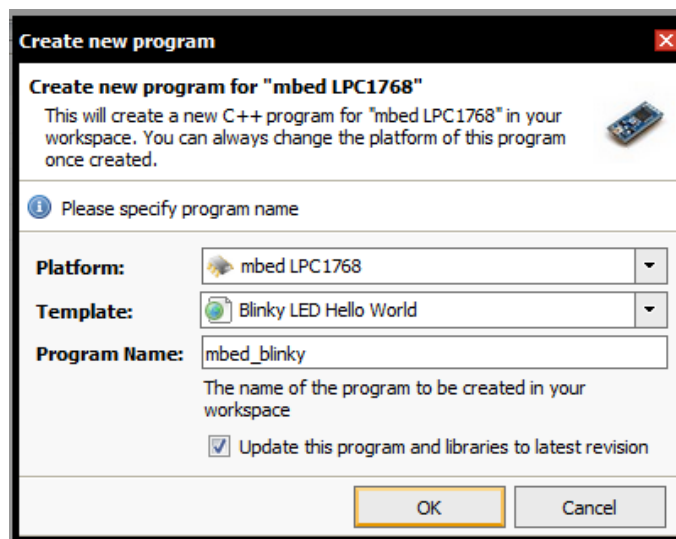


Figura 4.15 Ventana de opciones.

Para crear un *script* se emplea la pestaña *new* situada en la esquina superior izquierda de la pantalla de la figura 4.14b. Aparecen entonces las opciones a aplicar al nuevo programa, entre las que se encuentra el dispositivo en el que se va a ejecutar (*LPC1768* en este caso), si se quiere crear desde una plantilla para mayor comodidad a la hora de cargar librerías y el nombre del programa, tal y como se muestra en la figura 4.15

A continuación se puede acceder al editor de *scripts* para introducir el código que se ejecutará en el microcontrolador. Una vez terminado, basta con presionar el botón *compile* o el atajo `ctrl+D`. Si el proceso de compilación se realiza sin errores, se descargará un archivo en extensión `.c` en la computadora desde la que se accede al IDE. Para cargarlo en la *LPC1768*, simplemente se conecta ésta a través de USB y se copia el archivo recién compilado a la carpeta del dispositivo, como si se tratara de una memoria USB convencional. Para que el programa comience a compilar, basta con reinicializar la placa, bien desconectando

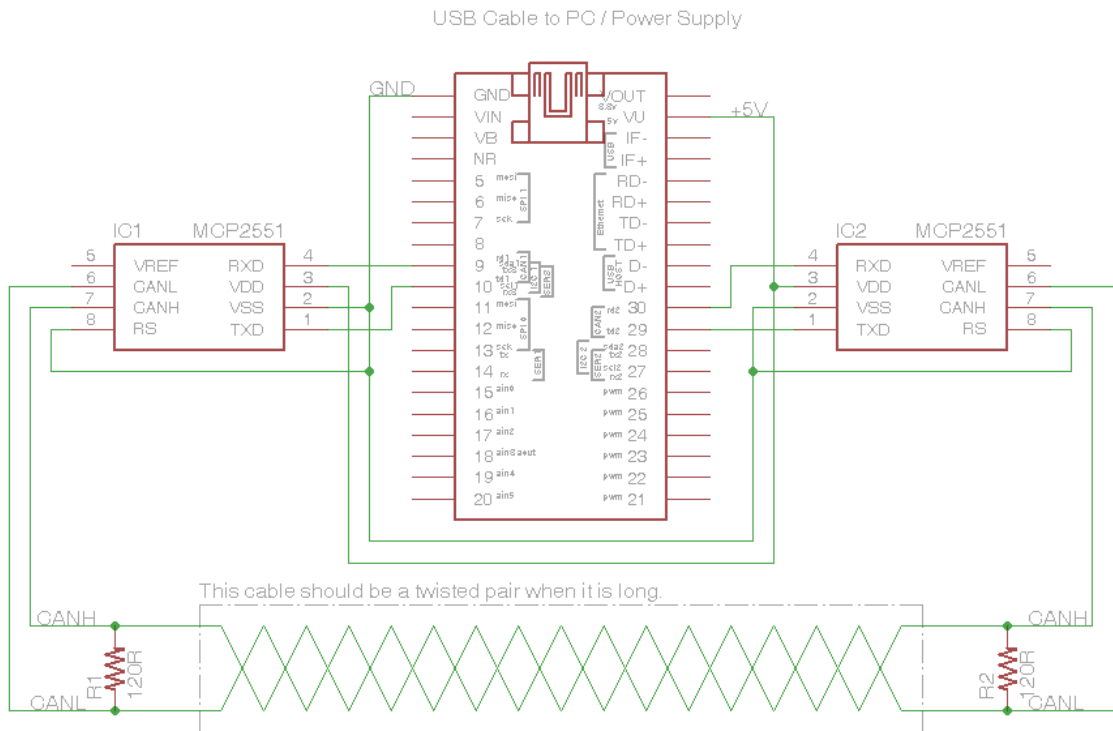


Figura 4.16 Esquema de la prueba [13].

la alimentación o pulsando el botón *Reset* que se encuentra sobre ella.

4.3.2 Comunicación entre canales

Aprovechando los dos canales CAN independientes que ofrece este equipo, en primer lugar se desarrolló una prueba de las comunicaciones CAN entre los puertos de la LPC1768. Los diagramas de conexión, así como toda la información necesaria para realizar el montaje se encontraron en una página del soporte en línea de *Mbed* <https://os.mbed.com/users/WiredHome/notebook/can---getting-started/>.

La manera de realizar las pruebas de comunicación entre los canales de la *Mbed* fue a través de la utilización de una placa de prototipos o *Protoboard*, ya que los terminales del dispositivo se encuentran espaciados para facilitar su integración con este tipo de elementos de conexión. La topología de estas placas consiste en una serie de receptáculos provistos de pinzas metálicas que presentan continuidad eléctrica a lo largo de una fila. En los bordes normalmente se incluyen dos columnas de conectores que recorren la placa en su dimensión más larga, que suelen usarse como raíles de alimentación. En el centro de la misma se encuentra un espacio que corta la continuidad de las filas, cuyas dimensiones están pensadas para alojar un paquete de circuito integrado de tipo DIP. Las distintas pistas pueden ser conectadas entre sí con latiguillos o *jumpers* para formar cualquier circuito eléctrico sin la necesidad de soldar sus componentes.

En el caso de la prueba realizada, en la figura 4.17 se presenta el *layout* de la *Protoboard*. La tensión de alimentación se obtiene directamente del pin etiquetado como VU (*5V USB out*) y se conecta a los raíles longitudinales de la placa, desde donde se puede conectar de manera conveniente a los pines número 3 de los MCP2551 (VDD). En cuanto a la masa del circuito, ésta viene proporcionada por el pin 1 del LPC1768 (0V), y se debe conectar al pin 2 de cada uno de los IC's. Con objeto de llevar la tierra a una posición más cómoda para conectarla a los pines 8 de los integrados (V_S), se hizo necesario realizar una conexión en puente entre los dos raíles longitudinales interiores.

A continuación se insertaron dos resistencias de 120Ω y $1/4W$ entre los extremos del bus (pines 6 y 7; CANH y CANL, respectivamente), y se proporcionaron los conectores que forman propiamente el bus, que unen dichos pines en ambos MCP2551. En el caso de distancias superiores al orden de 40m, se recomienda

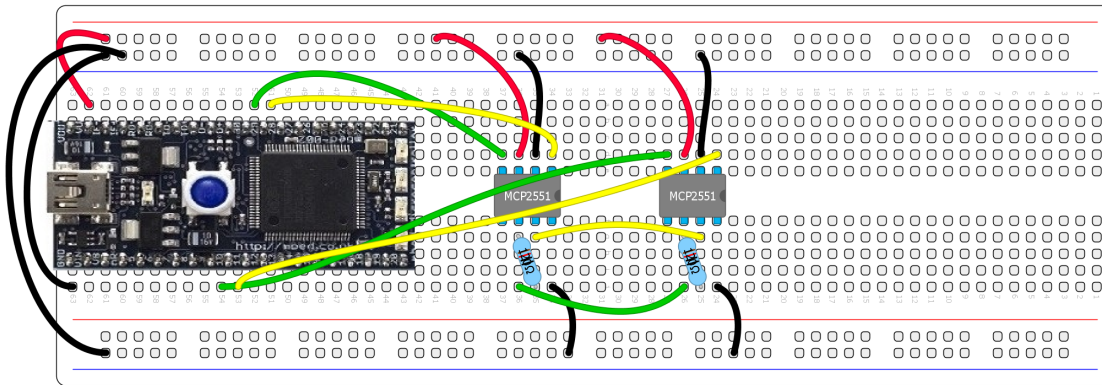


Figura 4.17 Protoboard con las conexiones necesarias.

además que estos dos cables se encuentren trenzados de manera compacta para evitar interferencias electromagnéticas. Además, cuanto mayor sea la longitud respecto a esta medida, la velocidad de comunicación se verá reducida gradualmente.

Por último se conectan de manera ordenada los dos canales a los transceptores; de un lado la pareja 9-10 del LPC1768 a la pareja 4-1 de un integrado, y por otra la pareja 29-30 a la 4-1 del restante. Es importante respetar este orden para conseguir que la prueba sea satisfactoria.

Comunicación serie

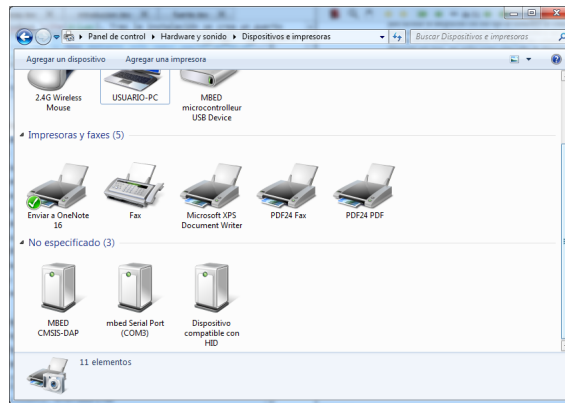
Es posible monitorizar la salida del código que se esté ejecutando en la *LPC1768* mediante la comunicación a través de puerto serie virtual, que se realiza utilizando el mismo terminal USB mediante el que se conecta al ordenador para cargar los programas. Para ello es necesario descargar los controladores en el caso de que se quiera visualizar en un ordenador con SO basado en Windows [14]. Tras la instalación se crea un puerto serie virtual desde el que acceder a la *LPC1768*. Los programas que se pueden utilizar para conseguir establecer un terminal con la Mbed mediante este nuevo puerto serie se encuentran listados en [15]:

- *Teraterm*
- *Putty*
- *Terminal* de Bray

En [1] se empleó *Putty* para la conexión a través de *SSH*, por lo que se aprovecharon los conocimientos adquiridos acerca de este *software*. En primer lugar se investigó el nombre del puerto serie en el que se encuentra conectada la Mbed, para lo cual se accede al administrador de dispositivos a través del panel de control de Windows, habiendo conectado previamente la Mbed al equipo. En la figura 4.18a se muestra que en este caso se trata de COM3. En la ventana principal de *Putty* (fig 4.18b) se accede a la opción de conexión por puerto serie y se introduce este nombre en el puerto a conectar. Los parámetros de conexión se pueden dejar por defecto, y son los siguientes:

- Velocidad (Baudios): 9600
- Bits de datos: 8
- Bits de parada: 1
- Paridad: Ninguna
- Control de flujo: XON/XOF

Pulsando sobre el botón *Open* debe abrirse un terminal en el que se muestran las salidas del programa que se esté ejecutando en la *LPC1768*. Para comprobar que la conexión es correcta, se realizó un pequeño programa que se muestra a continuación:



(a) Administrador de dispositivos .

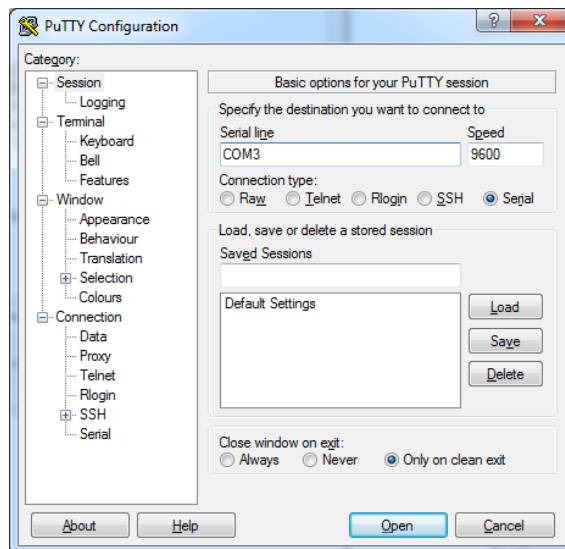
(b) Ventana principal de *Putty* .

Figura 4.18 Visualización a través de terminal.

Código 4.7 Programa de prueba.

```

1 #include "mbed.h"
2
3 int main() {
4 while(1) {
5 printf("hola mundo");
6 break;
7 }
8 }

```

Si se compila el programa tal y como se explicó en la sección 4.3, se copia el archivo *.bin* en la carpeta del dispositivo, se abre un terminal en *Putty* y se pulsa el botón *reset* de la placa, el terminal debe reflejar la frase "hola mundo", como se puede apreciar en la figura 4.19.

Un primer problema que se observó en el terminal de *Putty* fue la incorrecta alineación de las sucesivas líneas de salida del programa. Esto se puede arreglar en la pestaña de opciones *Terminal*, tal y como se muestra en la figura 4.20a

Para facilitar el trabajo, es posible almacenar todas las opciones que se han ido introduciendo (nombre del

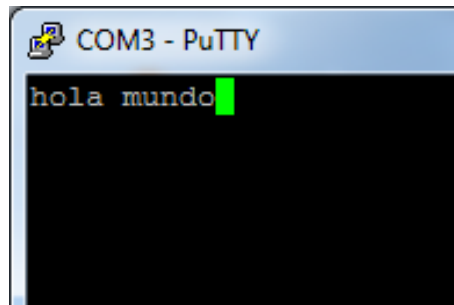
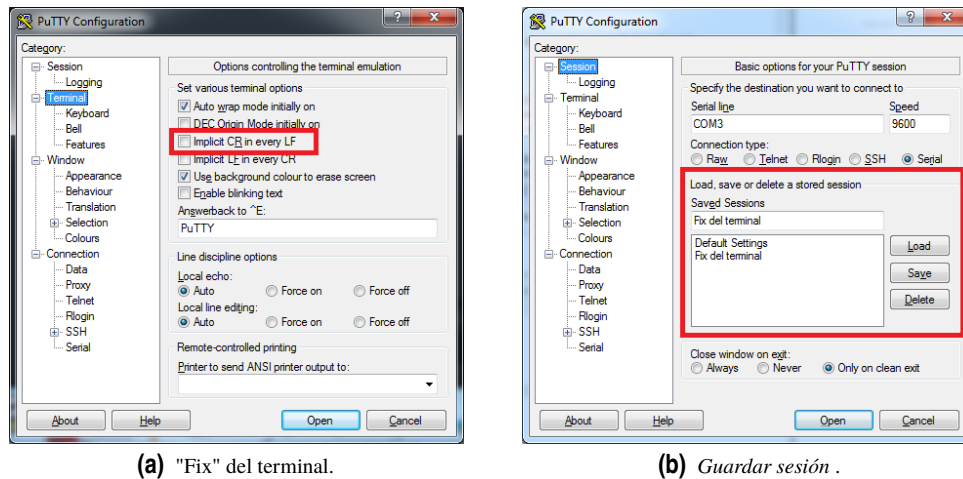


Figura 4.19 Mensaje de prueba en el terminal.



(a) "Fix" del terminal.

(b) Guardar sesión .

Figura 4.20 Mejoras de Putty.

puerto serie y opción para alinear terminal) en un perfil que se puede cargar automáticamente al inicio de Putty (obsérvese la figura 4.20b)

De esta manera se podrá tener una visión inmediata sobre la salida de la transmisión y la recepción de la LPC-1768 a través de la pantalla del PC del laboratorio, aspecto clave para la evaluación de la validez de la prueba.

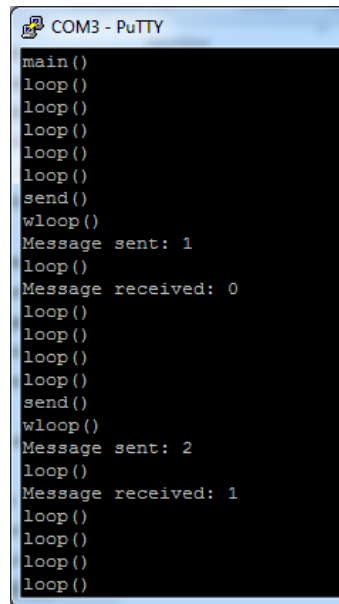
A continuación se analiza el código de ejemplo de intercomunicación de los canales propuesto en [13].

Código 4.8 Programa de prueba para comunicación CAN.

```

1 #include "mbed.h"
2
3 Ticker ticker;
4 DigitalOut led1(LED1);
5 DigitalOut led2(LED2);
6 CAN can1(p9, p10);
7 CAN can2(p30, p29);
8 char counter = 0;
9
10 void send() {
11 printf("send()\n");
12 if(can1.write(CANMessage(1337, &counter, 1))) {
13 printf("wloop()\n");
14 counter++;
15 printf("Message sent: %d\n", counter);

```



```

main ()
loop ()
loop ()
loop ()
loop ()
loop ()
loop ()
loop ()
send ()
wloop ()
Message sent: 1
loop ()
Message received: 0
loop ()
loop ()
loop ()
loop ()
loop ()
send ()
wloop ()
Message sent: 2
loop ()
Message received: 1
loop ()
loop ()
loop ()
loop ()

```

Figura 4.21 Salida del programa de intercomunicación del LPC-1768.

```

16 }
17 led1 = !led1;
18 }
19
20 int main() {
21 printf("main()\n");
22 ticker.attach(&send, 1);
23 CANMessage msg;
24 while(1) {
25 printf("loop()\n");
26 if(can2.read(msg)) {
27 printf("Message received: %d\n", msg.data[0]);
28 led2 = !led2;
29 }
30 wait(0.2);
31 }
32 }

```

En primer lugar se define un *Ticker*, que consiste en una interfaz para establecer una interrupción recurrente que llame a una función con una determinada frecuencia. A continuación se asignan las salidas que se emplean en la prueba; dos leds indicadores y las dos interfaces CAN. Asimismo se asigna el valor 0 a un contador necesario para la función *send()*. Ésta imprime por pantalla una línea cada vez que se ejecuta, y si se envía correctamente el mensaje CAN (con las siguientes características: identificador 1337, dirección señalada por la variable *counter* como contenido y longitud de 1 bit) al bus mediante la función *CANMessage* el contador se incrementa y notifica el éxito de la operación, cambiando de estado el led 1. Para la función principal se especifica que el *Ticker* produzca una interrupción para mandar un mensaje cada segundo, para crear después otro en blanco que será en el que se almacenen los sucesivos enviados por *send()*. Por último y de manera cíclica, si se tiene éxito leyendo con *can2.read*, se notifica y se cambia de estado el segundo led, produciéndose entonces una espera de 0.2s.

Con ello debe apreciarse un parpadeo sucesivo de los dos leds indicadores; el del canal 1 para la escritura y el del 2 para lectura. Esto se comprobó sin problemas, lo cual indica el correcto funcionamiento de la prueba y garantiza que se es capaz de leer y escribir en el bus de datos mediante la LPC-1768. Además, haciendo uso de la comunicación puerto serie, se pudo comprobar la correcta ejecución del programa, tal y como se muestra en la figura 4.21

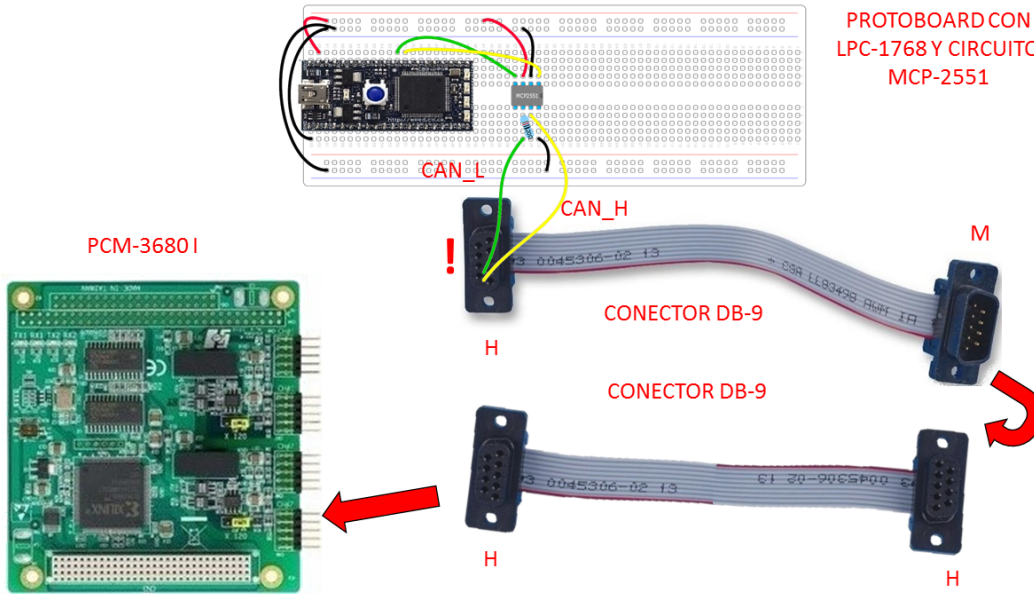


Figura 4.22 Diagrama de conexión entre placas.

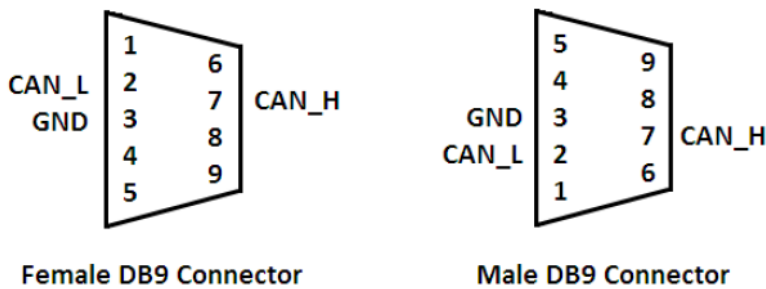


Figura 4.23 Pinout del conector DB9.

4.4 Comunicación entre dispositivos

Una vez probado cada uno por separado, se procedió a conectarlos entre ellos para comprobar las capacidades conjuntas del bus CAN y el sistema operativo en tiempo real implementado para la ECU. El esquema de conexión se muestra en la figura 4.22

Conviene hacer referencia al *pinout* del conector DB9 que comunica ambos dispositivos. En [10] se detallan las asignaciones de cada uno de ellos, que para una referencia rápida se encuentran en la figura 4.23. Es importante tener en cuenta si la terminación es macho o hembra para realizar las conexiones correctamente. Por otra parte, no es necesario conectar un jumper al terminal de tierra. Nótese también que se ha omitido del circuito uno de los traseptores MCP-2551, puesto que este componente ya se encuentra implementado en la placa *PCM-3680I*. Mediante este esquema de conexión se está en condiciones para comenzar con los dos casos que se consideraron. En las siguientes secciones se analiza el bus con la ECU como receptor (RX), siendo la *LPC-1768* el elemento complementario.

4.4.1 Prueba de funcionamiento

El programa para escribir en el bus CAN mediante la *Mbed LPC1768* se obtuvo de [16], y se adjunta a continuación en el código 4.9:

Código 4.9 Programa para escribir en el bus CAN.

```

1 #include "mbed.h"
2
3 DigitalOut led1(LED1);
4 CAN can1(p30, p29);
5 char counter = 0;
6 int main() {
7   can1.frequency(125000);
8   printf("send... ");
9   while(1) {
10    if (can1.write(CANMessage(1, &counter, 1))) {
11     printf("Message sent: %d\n", counter);
12     counter++;
13     led1=!led1;
14    }else{
15     can1.reset();
16    }
17    wait(1);
18   }
19  }

```

El funcionamiento del programa consiste en utilizar el canal que se establece entre los pines 29 y 30 de la *Mbed* para escribir un mensaje CAN cada segundo, en el se envía un contador en el campo de datos y se imprime por pantalla tanto el contenido del mensaje como el estado de la comunicación, además de emplear un led como indicador luminoso de cada transmisión. En la línea 10 se evalúa mediante un bucle *if* el éxito de la transmisión, *reseteando* el bus si no se produce.

Es de vital importancia para la comunicación que tanto RX y TX se encuentren a la misma frecuencia. En este caso hay que recordar que, según lo introducido en el código 4.4, el bus CAN transfiere los datos a 125000 bits por segundo. La manera de seleccionar la frecuencia CAN en la LPC1768 es mediante la asignación del atributo `frequency` al objeto `can`, línea 7. Si se omite este paso la comunicación no será posible en ningún sentido, corriéndose el riesgo de bloquear el sistema operativo.

En el caso de la Unidad de Control Electrónico, formada por la PCM-3365 e integrada en el bus mediante el módulo *PCM-3680I*, basta con utilizar el programa de ejemplo de lectura CAN para recibir la información. Las únicas modificaciones que se introdujeron fueron la eliminación de los caracteres "0x" que preceden a los bits contenido del mensaje (puesto que el programa de escritura de ejemplo envía números en formato hexadecimal) y de todos los mensajes que imprimen por pantalla el contenido salvo uno, puesto que el mensaje que se pretende enviar tiene una longitud de un bit. Además, por simplicidad, se omite la impresión del DLC o longitud en bits del mensaje.

El resultado que se debe visualizar en el *shell* del PCM-3365 puede deducirse del análisis del código de lectura de ejemplo, detallado a continuación con las modificaciones realizadas.

Código 4.10 Programa de ejemplo *readCAN*.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <net/if.h>
6 #include <sys/types.h>
7 #include <sys/socket.h>
8 #include <sys/ioctl.h>
9
10 #include <linux/can.h>

```



```
11 #include <linux/can/raw.h>
12 #include <linux/can/error.h>
13 #include <fcntl.h>
14
15 int main(int argc, char **argv)
16 {
17     int s,i;
18     int nbytes;
19     struct sockaddr_can addr;
20     struct can_frame frame;
21     struct ifreq ifr;
22     char ifname = argv[1];
23     int setflag,getflag,ret =0;
24     if (argc != 2)
25     {
26         printf("usage: readCAN <CAN interface> \n");
27         printf("ex. readCAN can0\n");
28         return 0;
29     }
30     if((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
31         perror("Error while opening socket");
32         return -1;
33     }
34
35     strcpy(ifr.ifr_name, ifname);
36     ioctl(s, SIOCGIFINDEX, &ifr);
37
38
39     addr.can_family = AF_CAN;
40     addr.can_ifindex = ifr.ifr_ifindex;
41
42     if(bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
43         perror("Error in socket bind");
44         return -2;
45     }
46
47
48     setflag = setflag|O_NONBLOCK;
49     ret = fcntl(s,F_SETFL,setflag);
50     getflag = fcntl(s,F_GETFL,0);
51
52     can_err_mask_t err_mask = CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF;
53     ret = setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, &err_mask, sizeof(
54         err_mask));
55     if (ret!=0)
56         printf("setsockopt fail \n");
57
58     printf("using %s to read\n", ifname);
59
60     while(1)
61     {
62         nbytes = read(s, &frame, sizeof(frame));
63         if (nbytes > 0)
64         {
65             if (frame.can_id & CAN_ERR_FLAG)
66                 printf("error frame\n");
67             else
```

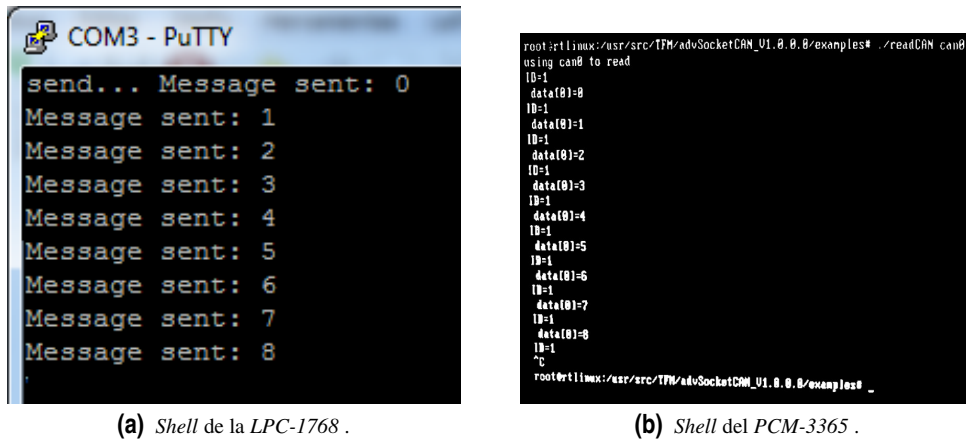


Figura 4.24 Salidas por pantalla de la primera prueba.

```

67     {
68         printf("\nID=%d \n",frame.can_id);
69         for (i =0;i<${frame.can\_dlc};i++)
70             {
71                 printf("data[%d]=%d \n",i,frame.data[i]);
72             }
73     }
74
75 }
76 }
77
78
79 return 0;
80 }

```

En la línea 68 se saca por pantalla el identificador del mensaje, mientras que en la 71 se imprime el contenido del mismo. La salida por pantalla va aumentando de línea con la frecuencia que se ha introducido en el código de escritura del bus en la *LPC-1768*, tal y como se muestra en la figura 4.24.

A continuación se procedió a someter a carga al sistema mientras se realizaba la comunicación, ejecutando la herramienta *hackbench* con los parámetros que se emplearon en la prueba del SO (sección 3.4.2). Además, se presionó el botón *reset* de la *LPC-1768* de manera aleatoria para comprobar la robustez en cuanto a ejecución del programa de lectura en la ECU ante interrupciones imprevistas. Los resultados fueron idénticos a los expuestos en la sección 3.4, lo cual resulta prometedor en la búsqueda del comportamiento determinista del sistema.

Así concluye este capítulo, habiéndose acondicionado un sistema de comunicaciones basado en el estándar CAN que resulta de aplicación directa en el ámbito del vehículo FOX. En el siguiente capítulo se empleará esta red junto con los demás dispositivos para realizar una prueba global de comportamiento de la unidad de control electrónico.

5 Montaje experimental y código de ejemplo para controlador de motor eléctrico

En este capítulo se emplean todos los conocimientos que se han ido desarrollando en las secciones anteriores como base para realizar un montaje experimental que sirva para demostrar que mediante el *hardware* y el *software* escogidos es posible disponer de un sistema de control electrónico del motor para el vehículo FOX que presente las características de determinismo que se fijaron como objetivo del presente proyecto.

Se comenzará introduciendo algunas nociones características de un código embarcado basado en *Linux*, describiendo funciones específicas de este tipo de sistemas operativos cuyas capacidades de tiempo real se añadieron mediante el procedimiento descrito en la sección 3.3, introduciendo a continuación el alcance de la prueba y sus componentes, finalizando por último con el código que se propone para la construcción del demostrador de concepto de unidad de control electrónica. Este análisis incluye la metodología empleada para llevar a cabo la tarea de control mediante código, con las elecciones realizadas y los problemas y soluciones halladas en el proceso de programación.

5.1 Procesos e hilos

En la presente sección se pretende exponer los elementos de programación más útiles para realizar la tarea del control electrónico del vehículo: procesos e hilos. La información presentada bajo estas líneas puede ampliarse consultando [17].

Tradicionalmente, un proceso UNIX correspondía a una instancia de un programa en ejecución, más precisamente a un espacio de dirección (*space adress*) y un conjunto de recursos dedicados a ejecutar el programa. Esta definición se vio formalizada por POSIX. El término hilo proviene del concepto de un hilo individual de ejecución, esto es, un camino "lineal" a través del código. POSIX define un hilo como los recursos necesarios para representar ese único camino de ejecución.

Históricamente siempre ha existido el concepto de proceso en los sistemas operativos UNIX. Cada proceso posee una serie de recursos asociados, incluyendo un espacio de dirección, registros de la CPU, un identificador o ID, una serie de descriptores de archivo abierto, un identificador de usuario, una pila o *stack*, ...etc. Mientras que éste es un buen modelo, sólo permite un camino secuencial de ejecución. Para conseguir algún grado de concurrencia en este modelo, es necesario crear múltiples procesos, que a menudo requieren algún tipo de comunicación entre procesos, que suele resultar cara y pesada en el contexto del tiempo de computación.

A finales de los años 80, el concepto de varios hilos de control se hizo popular en la comunidad de UNIX. La idea fundamental era crear un conjunto limitado de los recursos de un proceso y hacer varios juegos del mismo, permitiendo de esta manera la concurrencia dentro de un único proceso. Los recursos que se eligieron fueron los mínimos necesarios para representar un único estado de ejecución. Éste consistía principalmente en los registros de la CPU y la pila o *stack*. Cada juego se llamó entonces un "hilo". Esto permitió la concurrencia dentro de una aplicación sin necesidad de un mecanismo de comunicación entre hilos. Es importante notar que este modelo conservaba el concepto de proceso único, extendiendo la definición para incluir múltiples

hilos dentro de este proceso.

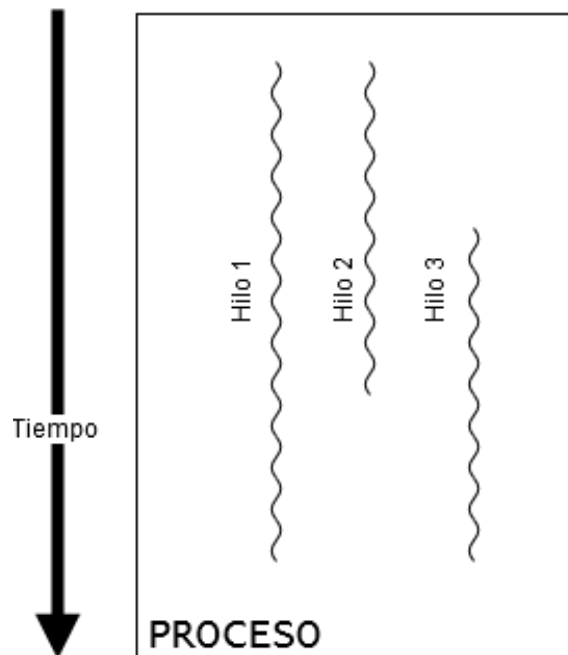


Figura 5.1 Esquema de procesos e hilos.

En esta época existieron muy pocas implementaciones experimentales de hilos, y ninguna de ellas en producción. Estaba claro, sin embargo, que resolvía una necesidad creciente, especialmente con el desarrollo de máquinas UNIX con procesador múltiple en el horizonte.

También durante este período la estandarización POSIX se encontraba poco desarrollada. Existía un gran impulso por crear un conjunto común de APIs que pudieran garantizarse para todas las implementaciones de UNIX. Algunos investigadores desarrollaron una API que implantaba el modelo multihilo y proponía incluirlo en el estándar POSIX. Se aceptó en forma de borrador bajo el epígrafe de extensiones de tiempo real. Aunque había poca experiencia en el mundo real con hilos, la intención era proveer un marco de trabajo común antes de que aparecieran múltiples implementaciones que competirían entre sí.

En los siguientes años, la API para hilos de POSIX (conocida como *threads*) ha pasado por numerosas revisiones y se incorporó al estándar en 1996. La mayoría, si no todas las implementaciones de UNIX incluyen una biblioteca de *pthread*, y hay muchas aplicaciones que las usan.

5.2 Montaje experimental del demostrador de concepto

Una vez presentado el método de programación se procede a diseñar un código de ejemplo que sirva como unidad de control electrónico para un pequeño motor eléctrico como demostrador de concepto. El montaje experimental respondió a los siguientes requisitos:

- La idea general consiste en emplear el *LPC 1768* como tarjeta de adquisición y escritura de señales tanto analógicas como digitales, comunicándose a través de bus CAN con el conjunto *PCM-3375* y *PCM-3680I*, que actúa como el núcleo de la unidad de control electrónica.
- El motor y su velocidad de giro se representan mediante un servomotor *Futaba S3003*, descrito en la siguiente sección.
- El control de revoluciones del motor se realiza mediante un potenciómetro lineal a modo de acelerador. Esta decisión resulta realista si se tiene en cuenta lo expuesto en [1], donde se aprecia que en el vehículo FOX el acelerador está compuesto esencialmente por un potenciómetro.

5.3 Programación del controlador

En líneas generales, el código principal de la ECU se programó mediante un hilo de ejecución POSIX, descrito en la primera sección del presente capítulo, asignando una alta prioridad para el *scheduler*.

5.3.1 Pthreads (POSIX threads)

La herramienta que se empleó en la programación consiste en hilos que cumplen con las especificaciones del estándar POSIX.

En primer lugar se presentan las funciones y la sintaxis de *pthread*. Es posible encontrar información más detallada de cada una de ellas en los manuales de Linux

Pthread_create

Permite crear un hilo POSIX. La sintaxis de esta función es la siguiente:

```
1 int pthread_create(pthread_t * thread,
2     const pthread_attr_t *attr,
3     void *(*start_routine) (void *),
4     void *arg);
```

- El primer argumento de la función es un puntero a una variable de tipo *pthread_t* que indica el identificador del hilo.
- El segundo argumento es un puntero a un puntero a una variable de tipo *const pthread_attr_t* con los atributos para la creación del hilo.
- A continuación se introduce un puntero a la función principal del hilo desde la cual se creará.
- Por último se encuentran los argumentos a la función del hilo.

Pthread_join

Recoge el retorno del hilo cuando éste termina. Para ello el mismo debe ser de tipo *joinable*.

```
1 int pthread_join(pthread_t thread,
2     void **retval);
```

- El primer argumento de entrada indica el identificador del hilo al que aplica esta función
- El segundo es un doble puntero vacío que apunta a la variable donde se almacena el retorno del hilo.

Pthread_exit

Esta función termina el hilo desde la que se llama.

```
1 void pthread_exit(void *retval);
```

El argumento de la función especifica el retorno que el hilo devolverá para *pthread_join*.

Pthread_cancel

La función *pthread_cancel()* manda una petición de cancelación al hilo desde la que se llama. Si el hilo reacciona o no, y cuándo ocurre la cancelación depende de dos atributos bajo el control de este hilo: su estado y tipo de cancelabilidad.

El estado de cancelabilidad de un hilo, determinado por *pthread_setcancelstate()*, puede ser *activado* (por defecto para hilos nuevos) o *desactivado*. Si el estado de cancelabilidad es *desactivado*, entonces cualquier petición de cancelación permanece en cola hasta que al hilo se le atribuya el estado *activado*. Si por el contrario un hilo tiene el estado *activado*, el tipo de cancelabilidad determina cuándo ésta ocurre.

El tipo de cancelabilidad, determinado por *pthread_setcanceltype*, puede ser asíncrono (*asynchronous*) o diferido (*deferred*) (por defecto). La cancelabilidad asíncrona quiere decir que el hilo puede ser cancelado en

cualquier momento (normalmente de forma inmediata, pero el sistema no lo garantiza). La cancelabilidad diferida significa que la cancelación del hilo se retrasará hasta que el hilo llame a una función que sea un llamado *punto de cancelación*. La lista de puntos de cancelación puede encontrarse en la página principal del manual de *Linux* para *pthread*, y tiene una extensión considerable [18].

Cuando actúa una petición de cancelación, los siguientes pasos ocurren para el hilo (en este orden):

1. Se crean los *handlers* de limpieza y se llaman.
2. Se llama a los destructores de datos específicos del hilo, en orden no especificado.
3. Finaliza el hilo.

Los pasos anteriores ocurren de manera asíncrona con respecto a la llamada de la función *pthread_cancel()*; el estado de retorno de *pthread_cancel* simplemente informa a quien la llama de que la petición de cancelación se ha puesto en cola con éxito.

Después de que el hilo cancelado haya terminado, su estado de retorno para *pthread_join* es *PTHREAD_CANCELED*.

```
1 int pthread_cancel(pthread_t thread);
```

A la hora de compilar es necesario adjuntar (*link*) la librería *pthread*. Para ello se ejecuta en la terminal del PCM-3365 el siguiente comando:

Código 5.1 Adjuntando la librería *pthread*.

```
1 root@rtlinux:~# cd /usr/src/TFM
2 root@rtlinux:~# gcc -pthread -o programa programa.c
```

donde *programa* y *programa.c* son el nombre del objetivo de la compilación y el script desde el que se compila, respectivamente.

Los elementos necesarios se detallan en la siguiente lista:

- Protoboard
- Cables de conexión
- Potenciómetro lineal de 10 K Ω
- Transceptor CAN *MCP 2551*
- Resistencia de 120 Ω terminadora de bus CAN
- *Mbed LPC-1768*
- Conjunto *PCM-3365* y *PCM-3680I*
- Servomotor *Futaba S3003*
- Componentes necesarios para el montaje de una fuente de alimentación de corriente continua de 5V @ 1.5A como máximo:
 - 4 diodos rectificadores 1N4007
 - Condensador de 2200 μ F y 25V
 - Regulador de tensión lineal *L7805*

5.3.2 Análisis del servomotor *Futaba S3003*Figura 5.2 Servomotor *Futaba S3003*.

Con objeto de emular el comportamiento del motor, se optó por emplear un pequeño servomotor de uso típico en aeromodelismo (figura 5.2). Sus características son las siguientes [19]:

Tabla 5.1 Características del servomotor *Futaba S3003*[19].

Alimentación	4.8-6 VDC
Velocidad de actuación	0.23s/60° @ 4.8 VDC; 0.19s/60° @ 6 VDC
Torque	3.2 kg*cm @ 4.8 VDC; 4.1 kg*cm @ 6 VDC
Señal de control	PWM (20ms)

El servomotor dispone de tres conductores: el de color negro se trata de la masa del dispositivo, y debe conectarse a la tierra del microcontrolador *LPC1768*; el de color rojo se emplea para la alimentación (4.8-6 VDC); por último el de color blanco sirve para enviar la señal de control al actuador del servo.

Respecto a ésta, se trata de una señal de tipo PWM (*Pulse Width Modulation*, o Modulación por ancho de pulso.) Este tipo de señales son de amplio uso en la actualidad, permitiendo el control y la alimentación de numerosos sistemas de manera eficiente. Se generan mediante un tren de pulsos con frecuencia fija, del cual es posible cambiar el ancho del pulso para regular la potencia transmitida.

En el caso del servomotor empleado en esta prueba, la señal a generar especificada por el fabricante posee una frecuencia de 50 Hz (período de 20 ms). Un ancho de pulso de 0.3 ms corresponde con una de las posiciones extremas, mientras que uno de 2.3 ms lo hace con la otra. Puede consultarse [20] para una guía comprensiva de las características y los métodos de conexión de este servomotor.

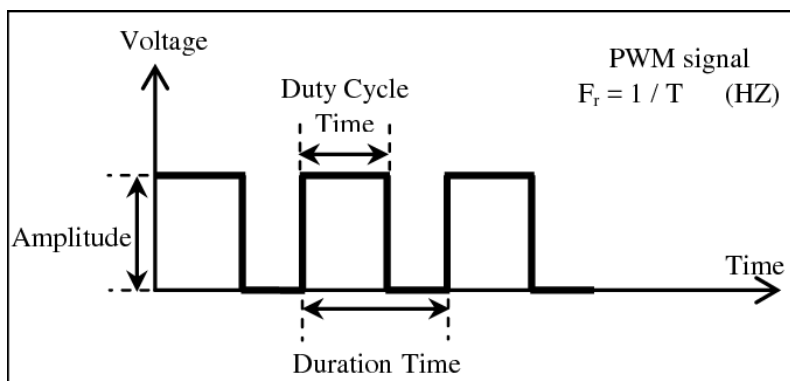


Figura 5.3 Señal PWM con sus características principales [21].

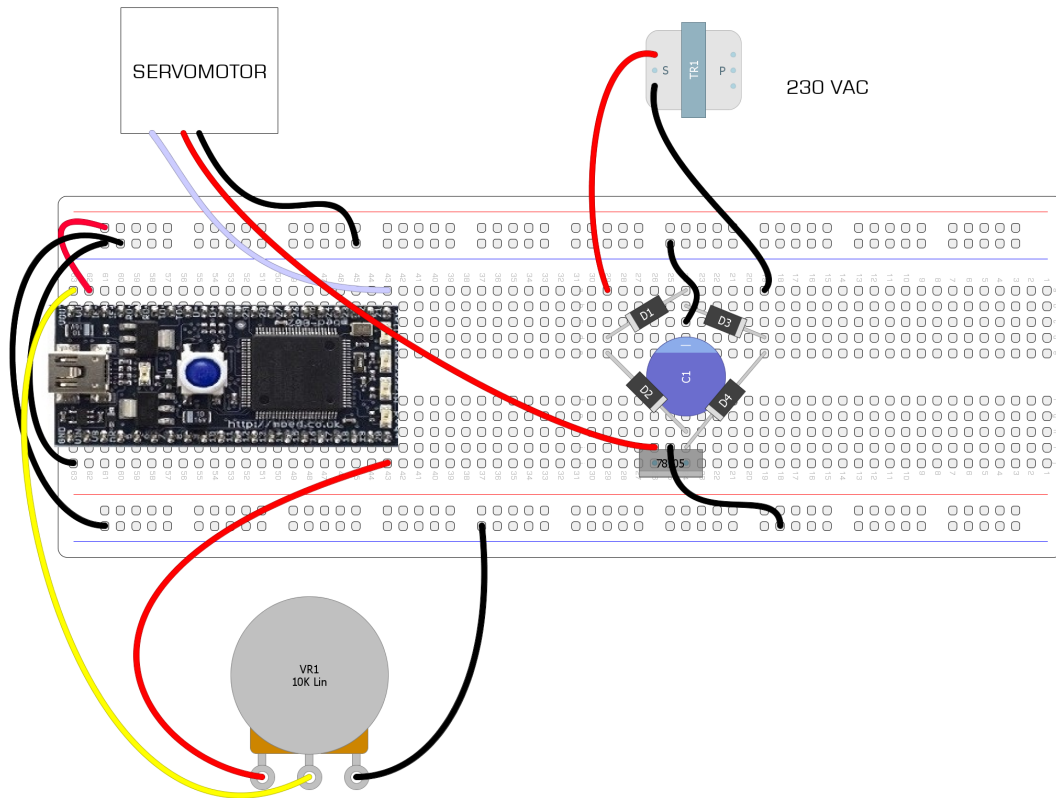


Figura 5.4 Montaje de prueba del servo.

Para probar que era posible controlar el servo mediante un potenciómetro se realizó el montaje recogido en la figura 5.4. El programa empleado se detalla en el código 5.2.

Código 5.2 Programa de prueba para el servomotor.

```

1 #include "mbed.h"
2 PwmOut servo(p21);
3 AnalogIn pote(p20);
4
5 int main() {
6     servo.period(0.020);           // periodo de la señal
7     while(1){
8         float potesc = (float)pote; // Typcasting de la variable para la posición del
           potenciometro.
9
10        servo.pulsewidth(0.002*potesc+0.0003); //Se asigna el ancho de pulso en función
           del potenciometro.
11    }
12 }

```

Del esquema de conexión cabe destacar que, a diferencia del abordado en el anexo A, dadas las características expuestas en la tabla 5.1, se hizo necesario utilizar un regulador lineal fijo con una salida de 5 VDC, siendo el resto del mismo similar al presentado en dicho anexo.

En cuanto al programa 5.2, en primer lugar se asignan los pines correspondientes a la salida PWM y a la entrada analógica de un potenciómetro; a continuación se asigna un periodo de 20 ms a la señal de control y

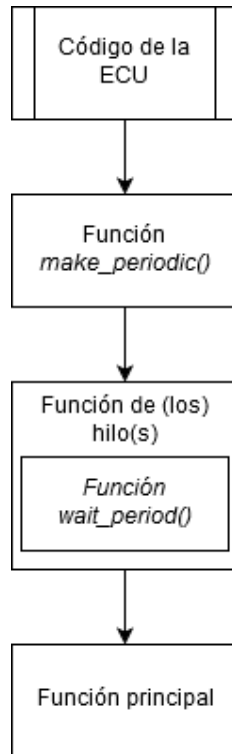


Figura 5.5 Diagrama de flujo de la función del controlador del vehículo.

por último se asigna el ancho de pulso necesario para el control de la posición. En este respecto es necesario tener en cuenta que la lectura de la posición del potenciómetro queda encuadrada en un rango de 0 a 1, que corresponde al valor máximo de voltaje entre el que oscila la lectura analógica (3.3 VDC). También hay que notar que el rango de variación del ancho de pulso está acotado entre 0.3 ms y 2.3 ms. Se trata entonces de encontrar una función que represente el ancho de pulso frente a la posición adimensionalizada del potenciómetro, de manera que en un extremo del mismo tome el valor de 0.3 ms y en el otro, 2.3 ms. La expresión es simplemente la de una recta de pendiente 2 ms y de ordenada en el origen 0.3 ms. La elección de una función lineal para representar esta correspondencia se justifica por la simplicidad de su expresión.

5.3.3 Análisis del código del controlador del vehículo

Se analiza ahora el código implementado en los dos dispositivos empleados.

Unidad de Control Electrónico (PCM-3365 y PCM-3680I)

En primer lugar se estudia el código para la unidad de control electrónico, formada por las placas *PCM-3365* y *PCM-3680I*, presentado en la sección C.1 del apéndice C. Para facilitar su comprensión, se presenta en la figura 5.5 un esquema de bloques explicativo.

Programación de tareas periódicas Para asegurar que el programa se ejecuta de manera determinista se programó el código para que las tareas de lectura, procesamiento y escritura se llevaran a cabo de manera periódica. Para ello se emplearon las funciones *make_periodic()* y *wait_period()*.

El elemento principal de programación en C++ empleado es un temporizador o *timer* que notifica su expiración en forma de *file descriptors*, cuya implementación se encuentra incluida en la función *timerfd_create*. Su sintaxis es la siguiente [22]:

```
1 \textit{timerfd\_create} (int clockid, int flags)}
```

El primer argumento de entrada indica el reloj del sistema que se pretende emplear. Los tipos disponibles son:

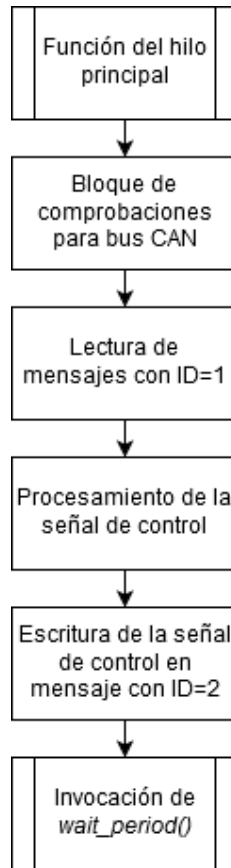


Figura 5.6 Diagrama de flujo de la función del hilo principal.

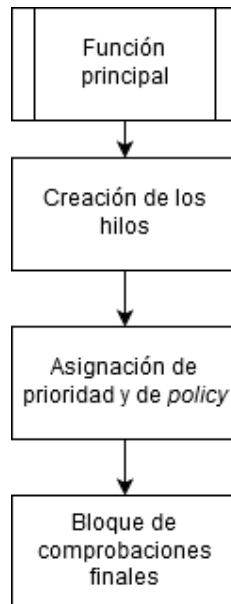


Figura 5.7 Diagrama de flujo de la función principal.

- **CLOCK_MONOTONIC**: Se trata de un reloj que se incrementa una cantidad fija de tiempo desde el inicio del sistema operativo.
- **CLOCK_REALTIME**: A diferencia del anterior, este reloj tiene la capacidad de ajustar los intervalos que va incrementando para ajustarse a la hora del sistema.

- `CLOCK_BOOTTIME`: Idéntico a `CLOCK_MONOTONIC`, pero es capaz de medir el tiempo durante el cual el sistema se encuentra en estado de suspensión.
- `CLOCK_REALTIME_ALARM`: Al igual que `CLOCK_REALTIME`, suma intervalos acordes con la hora del sistema, pero impide que quede en estado de suspensión.
- `CLOCK_BOOTTIME_ALARM`: De manera similar al anterior, impide que el sistema quede en estado de suspensión.

Debido a que no se planea que el sistema quede en estado de suspensión durante el funcionamiento del vehículo, se optó por emplear el reloj `CLOCK_MONOTONIC`, que es el que ofrece un mayor determinismo.

El segundo argumento es un parámetro para modificar el comportamiento de la función en cuanto a la gestión del *file descriptor*. En el presente estudio no se especificó ninguna (valor 0).

Antes del código de ambas funciones se crea una estructura con campos *timerfd* y *wakeups_missed*. En el primero se almacenarán los *file descriptors* que el *timer* genera cuando expira, y en el segundo se almacenará un entero que indicará las veces que el tiempo de ejecución del código del controlador supere al periodo que se fije para ello.

En *make_periodic* (línea 29 del código C.1) se comienza creando el *timer* y se asignan los atributos descritos en el párrafo anterior al puntero de estructura a la entrada *info*. A continuación se asignan los valores adecuados con el periodo deseado en la estructura *itimerspec* (inicializado en la línea 35). Estas asignaciones son necesarias para la función *timerfd_settime()*:

```
1 int timerfd_settime(int fd, int flags,
2     const struct itimerspec *new_value,
3     struct itimerspec *old_value);
```

En este caso el primer argumento es el nombre del *file descriptor* a generar, el segundo es una máscara para indicar si se quiere que el tiempo del *timer* sea absoluto o relativo, el tercero se trata de los nuevos valores para el *timer* y el último es útil si se quieren guardar los valores anteriores en un *file descriptor* diferente. En cuanto al tercer argumento, en él se indica tanto el tiempo inicial del *timer* como el periodo que transcurre entre expiraciones. Estos parámetros se introducen mediante la estructura *itimerspec*:

```
1 struct itimerspec{
2     struct timespec it_interval;
3     struct timespec it_value;
4     };
```

new_value.it_value especifica el tiempo inicial de la expiración del *timer*, en segundos y nanosegundos. Si se asignan valores distintos de cero a estos campos se arma el *timer*. En el caso de *new_value.it_interval*, si se le asignan valores distintos de cero, se especifica el periodo en segundos y nanosegundos para que el *timer* se repita tras la primera expiración. Ambos campos son a su vez estructuras de tipo *timespec* para especificar estos tiempos:

```
1 struct timespec{
2     time_t tv_sec;
3     long tv_nsec;
4     };
```

En cuanto a la función *wait_period* se emplea *read()*:

```
1 ssize_t read(int fd, void *buf, size_t count);}
```

Esta función intenta leer un número *count* de bytes del *file descriptor* *fd* en el *buffer* especificado por la variable de entrada *buf*. En el caso de que el campo de expiración del *file descriptor* se encuentre vacío esta función se bloquea, tal y como se especifica en la documentación de la función *timerfd_create()* [22]. Este comportamiento es el que permite la periodicidad en la ejecución del hilo con el periodo especificado en *make_periodic*. Además, en esta función se almacena el número de ciclos de periodo especificado que se ha excedido el tiempo de ejecución del código del hilo, lo cual resulta idóneo para caracterizar el comportamiento del sistema.

Función del hilo del controlador Se encuentra en primer lugar, ocupando desde la línea 74 hasta la 177 del código C.1. En la figura 5.6 se presenta el diagrama de flujo de la misma. Anteriormente se ha creado una variable de tipo *pthread_t* que corresponde al identificador del hilo (*h1*). Las acciones principales que realiza el programa controlador son la lectura del bus CAN de los mensajes con bit identificador adecuado, generación de la señal de control y escritura de un mensaje CAN con la señal, que será leída por el actuador correspondiente (*LPC1768*). Para la lectura CAN se comienza con un bloque de comprobaciones, tal y como se puede encontrar en [11]. Es importante notar que la interfaz CAN que se emplea para la comunicación (*can0* o *can1*) debe incluirse en la variable *ifname*, y que debe coincidir con aquella a la que está conectado el cable DB9 descrito en 4.4.

Tras comprobar que no se ha producido ningún error, se entra en un bucle principal que lee continuamente los mensajes con identificador 1 (los que se deben leer), dejando 2 para los que se escriben. A partir de la línea 142 se encuentra una serie de bucles *if* que evalúan la posición del potenciómetro y asignan las variables de encendido para los LEDs, que llegarán a la *LPC 1768* a través de los mensajes CAN con *id = 2*. Para el control del servomotor que representa la salida de la unidad de control electrónico se emplea el comando de la línea 166, en la que el dato del mensaje proveniente de la *mbed LPC-1768*, que contiene la posición del potenciómetro, se adjunta en el quinto byte del mensaje de salida, con identificador 2, para que se modifique la señal de control del servo.

En la línea 171 se encuentra una llamada a la función *usleep()* que sirve de *stopping point* para la función *pthread_cancel*. Por último se llama a *pthread_exit* para salir del bucle y que devuelva un valor vacío (NULL).

Función del hilo secundario Con el objeto de comprobar las funcionalidades de *pthread_cancel* y de establecer un entorno de prueba realista, el código C.1 implementa una parada de emergencia que funciona mediante entrada por pantalla. Un bucle *while* comprueba periódicamente el valor de una bandera, que cambia de estado en el momento en que se introduce el carácter de parada (que en este caso se eligió 5). Cuando se sale del bucle se realiza una llamada a *pthread_exit* para terminar el hilo 2.

Función principal (*main*) Desde esta función se crean los hilos, se les asigna prioridad y *scheduling policy* y se evalúa el estado de los mismos en cuanto se terminan de ejecutar (por parada de emergencia). En cualquier caso, una salida forzada del programa principal (con la combinación de teclas CTRL+C) en el terminal *bash* acaba también con cualquier hilo que se haya creado a través de la misma. Este comportamiento puede cambiarse mediante algunos parámetros de las funciones de *pthread*, pero se decidió dejarlo por seguridad durante las pruebas. El diagrama de flujo se presenta en la figura 5.7

En primer lugar se crean las variables para almacenar la *scheduling policy* de los dos hilos, y su prioridad. En este ejemplo de prueba se decidió emplear valores bajos para no comprometer el correcto funcionamiento del equipo en caso de producirse un error a la hora de programar. En el caso de la *policy* FIFO, el valor máximo es de 99. El primer hilo en crearse es el del programa del controlador, tras lo cual se asigna la *policy* y la prioridad. Se procede de igual manera para el segundo bucle. Se emplea *pthread_join* para comprobar que los dos hilos han finalizado y poder continuar con la ejecución de la función principal.

Por último se incluye código para imprimir por pantalla tanto la *policy* como la prioridad de ambos hilos, así como el éxito a la hora de finalizarlos. Este bloque se ejecutará únicamente cuando se realice una parada de emergencia del controlador según lo anteriormente dispuesto.



Figura 5.8 Diagrama de flujo del código en la *Mbed LPC1768*.

Código de la tarjeta de adquisición de datos y actuadora (*Mbed LPC1768*)

El código implementado en la tarjeta *Mbed LPC1768* queda recogido en el código C.2 del apéndice C. Éste se compiló de la manera descrita en 4.3.1 en la tarjeta *Mbed LPC1768*. A continuación se explican los principales aspectos que presenta, adjuntándose el diagrama de flujo de la figura 5.8 para su mejor seguimiento.

En primer lugar, se inicializan todas las entradas y salidas del microcontrolador, esto es:

- Salidas digitales para los LEDs de a bordo.
- Salida de tipo PWM para la señal de control del servo.
- Entrada analógica para la posición del potenciómetro.

Además, en la línea 19 se inicializa la interfaz CAN para permitir la comunicación entre dispositivos. Todas las anteriores acciones han de realizarse respetando el *pinout* de la *Mbed LPC-1768*, presentado en la figura 4.12b.

la función principal comienza con la asignación de la frecuencia para la señal PWM de control del servo. Es importante su declaración antes del bucle *while* de la línea 28, puesto que de manera contraria no se produce un comportamiento correcto del dispositivo. Otro de los aspectos más importantes del código es la asignación de la frecuencia de transmisión de datos a través del bus CAN, presentada en la línea 30.

Una vez leídos los mensajes procedentes de la ECU se procede a realizar la actuación de control sobre los dos tipos de elementos configurados para ello: de un lado, un patrón de luces LED y de otro el servomotor descrito en la figura 5.2.

5.4 Caracterización del comportamiento

Cyclictest y *Hackbench*

Con objeto de asegurar el comportamiento determinista de todo el sistema, se realizaron todas las pruebas de idéntica manera a como se expone en la sección 3.4.2.

Un problema que surgió a la hora de proceder fue que la parada de la función principal con objeto de transferirla a segundo plano (tómese el código 3.9 como referencia) termina con los hilos que se están ejecutando a partir de la misma (*main*), dejando al sistema sin controlador. Para solucionarlo se hizo uso del comando *shell* "*nohup*, del inglés *no hung up*), cuya sintaxis para pasar la ejecución del controlador a segundo plano dejando libre el *shell* se presenta a continuación en el código 5.3

Código 5.3 Comandos *shell* para ejecutar el programa del controlador en segundo plano.

```
1 root@rtlinux:~# gcc -pthread -o programa programa.c && nohup ./programa>/dev/
  null 2>&1 &
```

En esta línea de comando *shell*, en primer lugar se compila el código *programa.c*, adjuntando la librería *pthread*, para a continuación ejecutar el programa directamente en el *background* (las dos operaciones se ejecutan secuencialmente empleando el operador *&&*). Esto libera el terminal para poder ejecutar el perfil de carga con *hackbench* y monitorizar la latencia con *cyclictest*. Nótese la sintaxis de la parte del comando que comienza con *nohup*; lo que se consigue con ella es redirigir la salida que se crea mediante este comando a */dev/null*, esto es, eliminándola por completo. En caso de no realizar este ajuste se creará un fichero de salida que irá incrementando su tamaño a medida que avanza el tiempo de ejecución del programa, llegando a llenar efectivamente toda la memoria de la partición principal, dejando inservible el computador.

Respecto a la latencia, seguidamente en la figura 5.9 se presenta el histograma obtenido de la ejecución del programa del controlador con un perfil de carga.

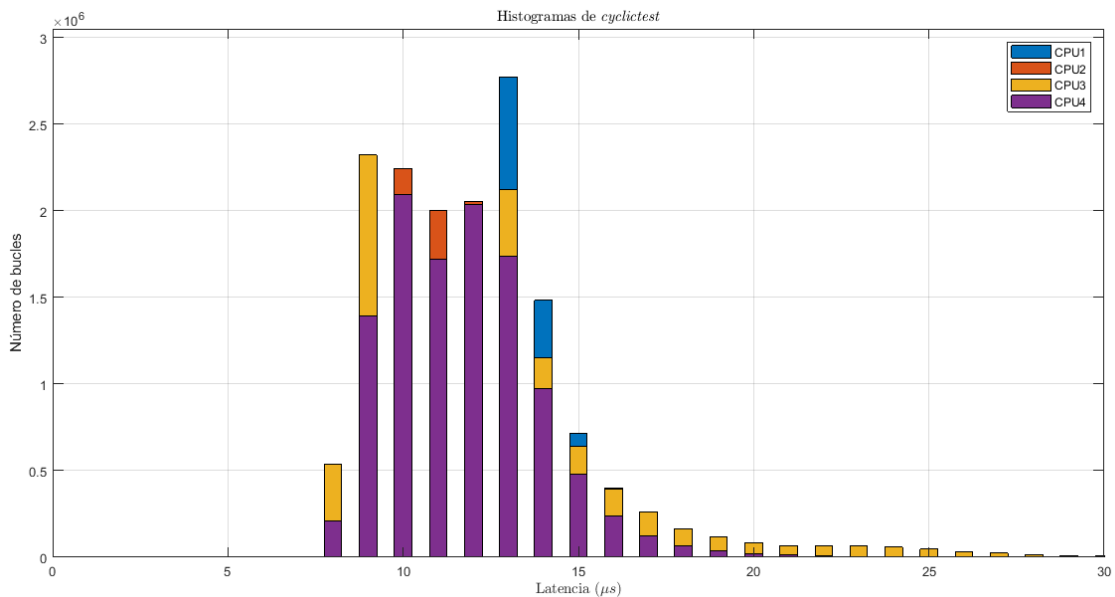


Figura 5.9 Histograma de latencias del montaje completo.

Se observa que los resultados en cuanto a la latencia son ligeramente superiores que en el caso contemplado en la sección de caracterización del comportamiento del sistema sin ejecutar el controlador (secc. 3.4). No obstante, la máxima latencia medida por *cyclictest* es de $71 \mu\text{s}$, muy por debajo de los $500 \mu\text{s}$ obtenidos en [1, Sección 6.6.3]. Nótese también cómo la distribución del histograma se traslada hacia latencias superiores comparado con el de las figuras 3.11 y 3.12. Este hecho puede interpretarse como indicativo de la carga que el montaje experimental supone en el sistema de control electrónico.

Medición del tiempo entre paquetes recibidos

La primera concepción del código principal de la unidad de control electrónica (código C.1) no consideraba la posibilidad de que la ejecución se realizara de manera periódica, sino que sirvió para comprobar que el programa respondía a estímulos externos en tiempo real. No obstante, tal y como efectivamente se ha presentado en dicho código y se ha explicado en detalle en la anterior sección, fue necesario estudiar un

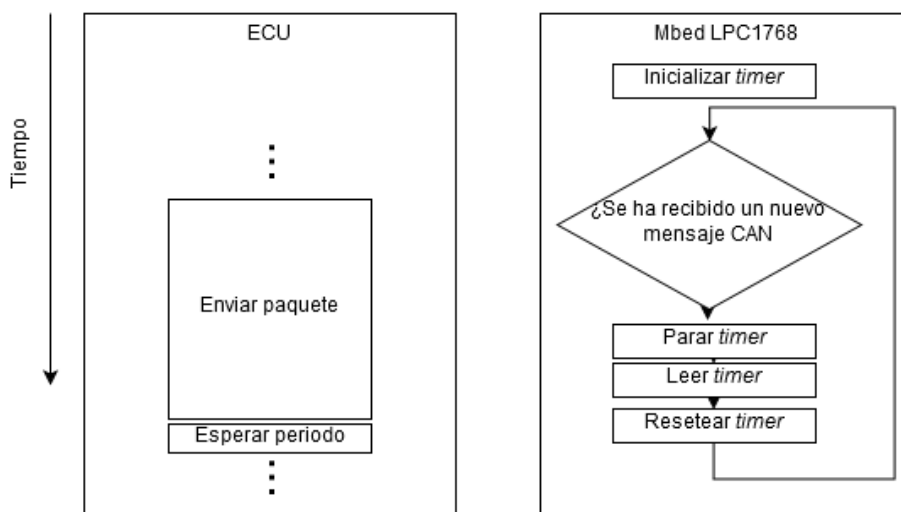


Figura 5.10 Diagrama de flujo de los dos dispositivos durante la prueba.

método para que se forzara al sistema a trabajar de manera periódica, de forma que se cumplieran los objetivos de velocidad de procesamiento. Esta filosofía de programación periódica permite tener la certeza de que el sistema se comporta de manera determinista en cuanto a tiempos de ejecución, lo cual constituye el objetivo más fundamental del presente proyecto más allá de la velocidad efectiva del sistema.

Para terminar de confirmar este comportamiento, se realizó una prueba que consistió en medir el tiempo entre que se reciben dos mensajes consecutivos en la *Mbed LPC1768*, obteniendo de esta manera una idea de la latencia del sistema global (bus CAN, y unidad de control principal *PCM-3365*). Para ello se emplearon varias herramientas de programación y de generación de archivos en el lenguaje C++.

Se creó un código de prueba previa implementación en el programa principal de la ECU en la tarjeta *Mbed LPC1768*, recogido en el código C.3 del apéndice C. El objeto de este programa no es otro que el de medir los tiempos entre recepción de paquetes a través del bus CAN.

El primer paso consistió en averiguar cómo obtener el tiempo de ejecución dentro de una función en el lenguaje *mbed*. Para ello es posible emplear uno de los *timers* de los que dispone, que proporciona en segundos el tiempo entre llamadas.

Su uso consiste en la declaración previa a la ejecución de la función principal de una variable de tipo *timer* (línea 17 del código C.3). Ya dentro de la función principal se invoca a `t.start()` para comenzar el *timer* y a `t.stop()` para detenerlo.

El siguiente problema consiste en realizar las medidas únicamente cuando se reciba un mensaje nuevo a través del bus CAN, como queda reflejado en la figura 5.10. La documentación del API del bus CAN proporcionada por *Mbed* [23] indica que la clase CAN posee una función atributo llamada *attach()*, que permite crear una interrupción cuando se produce uno de los siguientes eventos del bus [24]:

- CAN::RxIrq para nuevo mensaje recibido
- CAN::TxIrq para un nuevo mensaje transmitido o abortado
- CAN::EwIrq para una advertencia de error
- CAN::DoIrq para sobreescritura de datos
- CAN::WuIrq para estado de *wake up*
- CAN::EpIrq para error pasivo
- CAN::AllIrq para proceso de arbitraje perdido
- CAN::BeIrq para error del bus

Para la aplicación que se pretende llevar a cabo, la primera de ellas resulta idónea, puesto que en virtud de la documentación, es posible lanzar una función cada vez que se produzca este estado en el bus, esto es, que se reciba un nuevo mensaje. Basta entonces con programar dicha función con la parada, lectura y puesta a cero del *timer* anteriormente descrito.

El problema que surgió de esta implementación responde a la naturaleza de las interrupciones que se generan con *can.attach()*, que son de tipo ISR (*Interruption Service Routine*). Estas interrupciones presentan la particularidad, entre otras, de que no pueden implementar funciones de entrada/salida con *buffer*, tales como *printf()* o *fprintf()* para interrupciones con un tiempo similar al de vaciado de tal *buffer* sin que presenten fallos en la ejecución. Dichos tiempos son del orden de 1 ms.

Este comportamiento se apreció en el hecho de que periodos entre transmisión de paquetes mediante la *PCM-3680I* cada vez más bajos empezaban a arrojar unos valores de tiempo transcurrido por pantalla inconsistentes, llegando a quedar en el entorno de decenas de microsegundos cuando se fijaba el periodo en cualquier valor por debajo de 5ms.

Esto puede resolverse de varias formas, tal cual se recoge en [25]. No obstante, en [26] se proporciona una solución sencilla que consiste en la creación de una variable de tipo booleano que cambia a estado verdadero cada vez que se produce la interrupción. Esta variable se evalúa continuamente en la función principal, y si resulta ser verdadera, imprime por pantalla el valor de la lectura del *timer* que se ha almacenado en la línea 33 del código C.3.

El almacenamiento de los tiempos se terminó realizando de forma similar a la llevada a cabo en la sección 4.3.2: la salida por pantalla del *script* presentado en el código C.3 se visualiza mediante el programa *Putty*, proporcionando al mismo la opción de guardar todas las líneas de salida en un archivo con extensión *.log*. A partir de las medidas es posible crear un histograma y representarlo de forma similar a la presentada en el apéndice B.2.

Para facilitar el uso futuro se guardó la sesión al igual que en la sección 4.3.2, incluyendo todas las opciones correspondientes a la sesión del *fix* del terminal. En la figura 5.11 se presentan las principales opciones para llevar a cabo este último paso.

El código que se ejecuta en la ECU aparece recogido en el código C.4, empleado para para enviar mensajes CAN de manera periódica, siendo el tiempo entre ellos (periodo) y su número total dos variables que se pueden modificar a voluntad.

El funcionamiento del programa es idéntico al expuesto en el apartado del código C.1, con la creación de un *timer* que genera un *file descriptor* que bloquea la ejecución del hilo hasta que no se lee mediante la función *wait_period()*. En este caso, sin embargo, no se leen mensajes CAN ni se procesa su contenido, sino que simplemente se mandan a una tasa conocida. La diferencia principal de este código frente al C.1 es, como se ha comentado, la capacidad de controlar el número de mensajes CAN que se envían y el periodo de transmisión.

Para finalizar este apartado se adjuntan distintos gráficos en los que se presenta la información obtenida de esta última prueba de caracterización. Éstas se han obtenido fijando un periodo de 20ms (50 Hz), y un total de 100000 mensajes, lo cual sugiere un tiempo total aproximado de ejecución de 30 min. Al igual que en las demás pruebas, se ejecuta simultáneamente en segundo plano el programa de carga *hackbench* con el perfil habitual (véase el código 3.4.2).

En la gráfica 5.12 se puede apreciar el histograma de los tiempos de ejecución. Los intervalos en los que se produce el conteo tienen un ancho de $1\mu s$, y responden a una distribución similar a una de tipo normal. La tendencia que se observa es que se cumple el periodo establecido de 20 ms, con algún tiempo de margen en el sentido más favorable.

Asimismo, en la figura 5.13 se puede observar un detalle de la distribución completa de tiempos de ejecución (gráfica mensaje frente a tiempo de distancia con el anterior). Puede comprobarse la aleatoriedad de tiempos, aunque en límites razonables alrededor del periodo fijado. El hecho de que no se muestre para

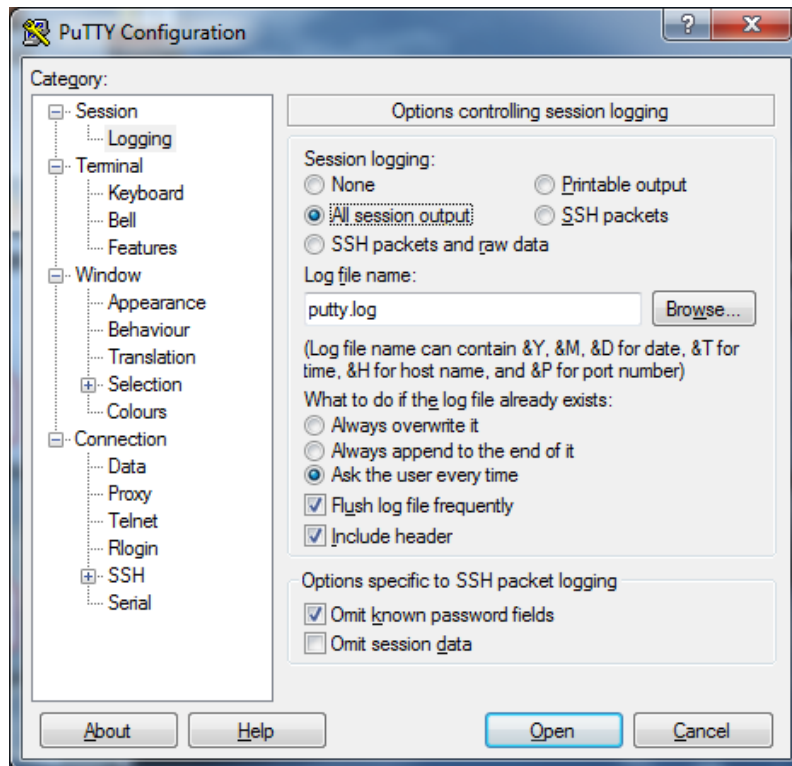


Figura 5.11 Opciones para crear un *log file* en Putty.

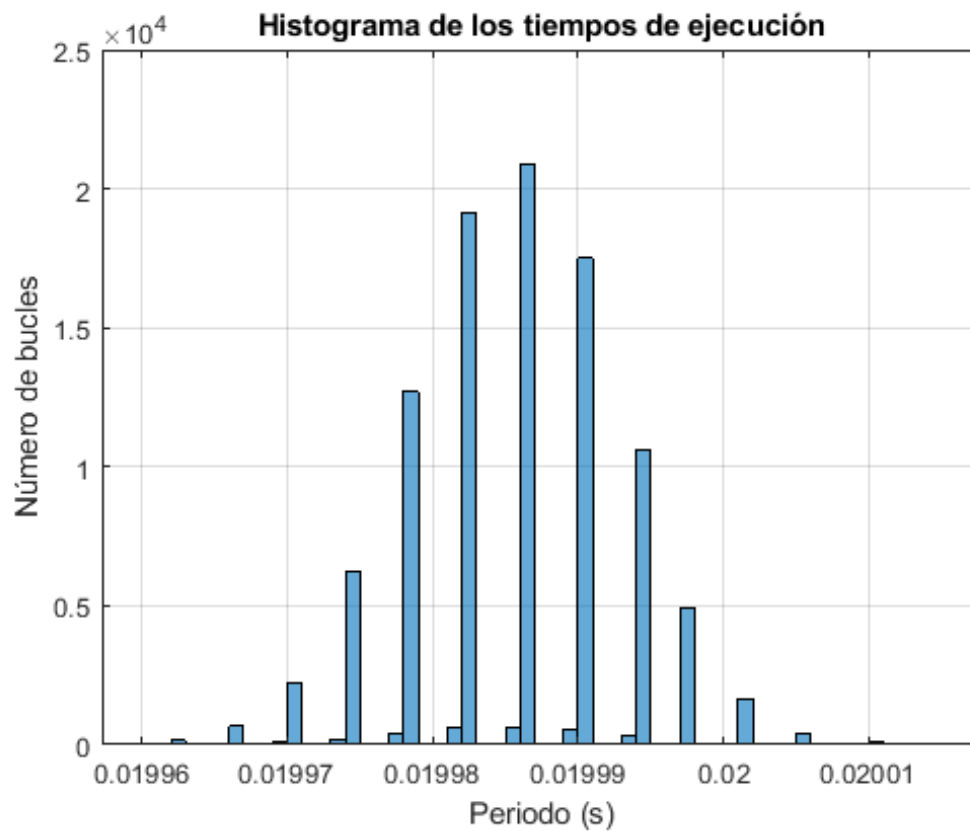


Figura 5.12 Histograma de tiempos entre mensajes.

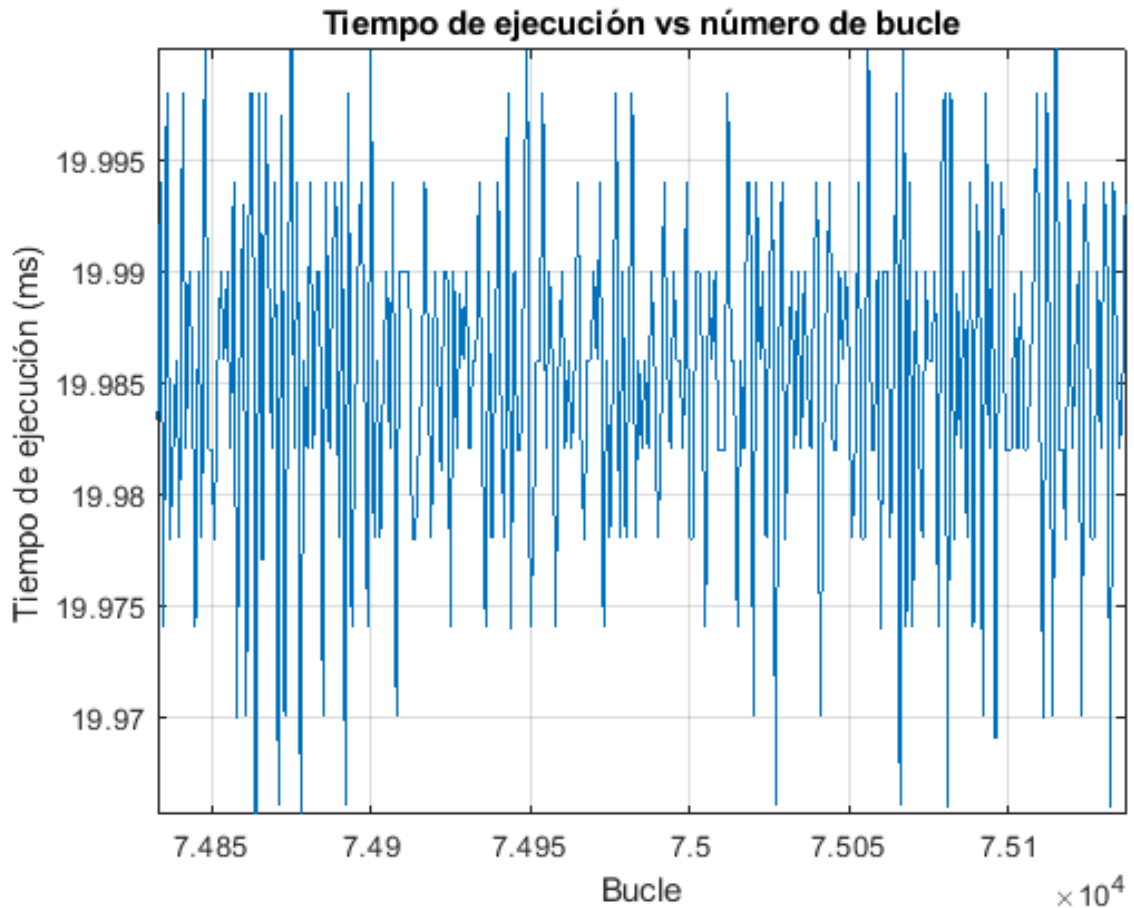


Figura 5.13 Distribución parcial de tiempos de ejecución.

todos los mensajes responde a la densidad de información que ello supondría.

Con esta sección de estudio de comportamiento, con los resultados favorables obtenidos, se alcanzan los objetivos que se marcaron para la realización de este proyecto. En la siguiente sección se recapitulan y analizan los resultados obtenidos durante el mismo.

6 Conclusiones y trabajo futuro

En este último capítulo se recogen los resultados principales del estudio llevado a cabo en este trabajo, y se valora si son adecuados o no para cumplir con los objetivos planteados al comienzo del proyecto.

6.1 Recapitulación y conclusiones

Mediante lo estudiado en los capítulos del bloque uno se ha conseguido poner a punto el computador *PCM-3365*. Esto ha incluido el *hardware* necesario (sección 2.4), así como todo el *software* para el sistema operativo *Slackware* de *Linux*, y su adaptación para operar bajo condiciones de tiempo real estricto. Las pruebas llevadas a cabo en la sección 3.4, en condiciones de carga relativamente exigentes comparadas con las que experimenta la unidad de control electrónico en el vehículo FOX muestran la idoneidad del sistema para su uso en dicha aplicación.

En cuanto al segundo bloque, se consiguió instalar con éxito una tarjeta de bus CAN (*PCM-3680I*) en el *PCM-3365*, junto con sus *drivers* correspondientes. El uso de un microcontrolador externo con capacidad de comunicación CAN (*LPC-1768*) ha resultado de utilidad en un doble propósito; de un lado, entender el funcionamiento a bajo nivel del protocolo, mediante el uso del circuito integrado del transceptor CAN *MCP2551*, y por otro, comprobar la comunicación entre dos dispositivos distintos.

El último bloque integró todos los anteriores dispositivos probados junto con otro nuevo, un elemento sobre el que actuar como es un servomotor. Ello ha servido para introducir la generación de una señal PWM y el uso de las entradas analógicas de un microcontrolador genérico.

Varias han sido las pruebas realizadas en este contexto; de un lado se implementó un montaje experimental en la sección 5.2 cuyos resultados controlando un servomotor fueron satisfactorios; de otro, se realizaron la caracterización del sistema completo como en la sección 3.4 y un seguimiento exhaustivo de los tiempos de ejecución a través del bus CAN, demostrando un comportamiento periódico (véase la sección 5.3.3), con el periodo establecido en los objetivos de la sección 1.3, y asegurando un éxito de transmisión absoluto de los mensajes incluso con cargas severas.

Con todo ello es posible concluir que se está en condiciones de migrar el código de los controladores que se encuentran actualmente implantados en el vehículo al nuevo sistema, y realizar pruebas de caracterización de su funcionamiento mientras se ejecutan dichos códigos. Este planteamiento se desarrolla a continuación en la sección dedicada a trabajo futuro.

6.2 Trabajo futuro

6.2.1 Pruebas de funcionamiento

El carácter experimental del sistema estudiado en el presente trabajo, si bien posee unas características que resultan bastantes prometedoras como se acaba de comentar, hace desaconsejable la instalación directa en

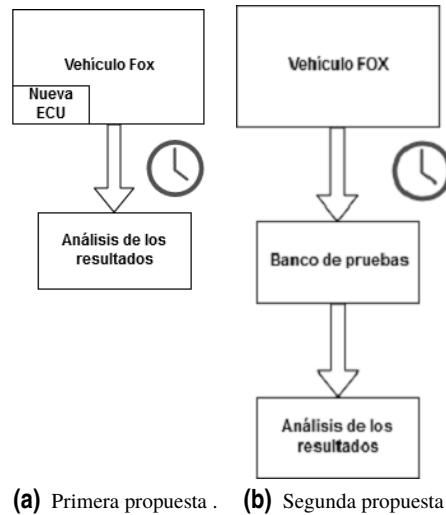


Figura 6.1 Esquemas de pruebas propuestos.

el vehículo para pruebas en vivo, debido a la situación de riesgo que podría suponer para el piloto y sus alrededores la pérdida efectiva de control del mismo. La línea de acción más adecuada consistiría en realizar un banco de pruebas externo ubicado en el laboratorio del FCCL, de manera que se pudiera conocer la respuesta del sistema a los datos recogidos de los distintos sensores en la unidad de control electrónico y observar la acción de control sobre los actuadores del vehículo, en este caso los motores. Este tipo de pruebas reciben en el contexto de la validación de conceptos operacionales en ATM (*Air Traffic Management*)[27] el nombre de *Pruebas en modo sombra*, y suponen el nivel de desarrollo del proyecto más alto sin realizar pruebas "en vivo", en las que el vehículo sería directamente controlado por el nuevo sistema.

Para llevar a cabo dichas pruebas, se plantean dos métodos hipotéticos. Los esquemas de los mismos se presentan en la figura 6.1:

- El primero consistiría en emplazar el nuevo sistema en el interior del vehículo, conectando las entradas de los sensores, y programándolo de manera que las acciones de control se guardaran en un fichero para poderlo consultar a posteriori.
- El segundo consistiría en guardar la señal de actuación y los tiempos de ejecución de la misma durante el funcionamiento del vehículo en un fichero que podría alimentar al banco de pruebas externo para comprobar el correcto funcionamiento

Apéndice A

Prueba de funcionamiento de PCM-3910

En el presente anexo se describe de forma breve la manera de probar el funcionamiento del sistema de control del vehículo sin emplear una fuente de alimentación ATX, construyendo para ello un pequeño convertidor AC/DC de 18V para simular las baterías. La elección de este nivel particular de voltaje se eligió por conveniencia, y se puede modificar sin más que cambiar un único componente del circuito propuesto. Como se ha comentado, el esquema de conexión es parecido al expuesto en la figura 2.3b del capítulo 2, sólo que en lugar de las baterías se conecta la salida de este circuito:

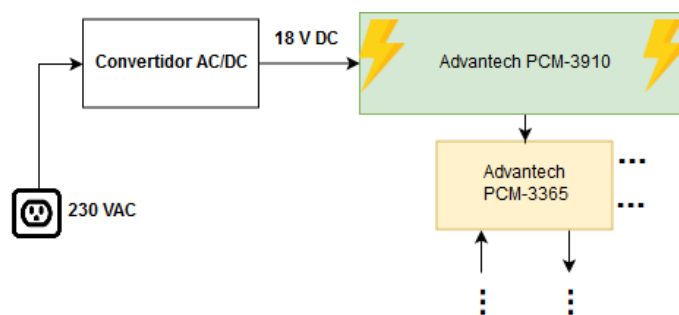


Figura A.1 Esquema con convertidor.

El tipo de fuente que se construyó es del tipo lineal regulada, en las que no existe tratamiento digital de la tensión para su conversión. La desventaja de este tipo de circuitos es su baja eficiencia frente a las modernas fuentes conmutadas (SMPS, *Switched Mode Power Supply*), que utilizan modulación de pulsos (PWM). Ello se hace patente en el tamaño de los componentes necesarios y en las pérdidas por calor en algunos de sus componentes. No obstante, su construcción es sencilla si se tienen en cuenta algunos aspectos que se tratan a continuación.

A.1 Consideraciones de diseño

En primer lugar se recurrió al *datasheet* de la *PCM-3910*, en el que se encontró que la tensión de entrada admisible se encuentra entre 10VDC y 24VDC. Los bloques básicos del circuito son *transformador, rectificador, filtro y regulador*

El transformador elegido dispone de una entrada de 230VAC y una salida de 24VAC (figura A.2a). Para convertir este nivel a corriente continua es necesario un rectificador, compuesto por cuatro diodos (puente de onda completa). La tensión que se obtiene tras este componente presenta un fuerte rizado, para lo cual se añadió un filtro consistente en un condensador de una capacidad adecuada. Es importante notar que en la rectificación en un puente de onda completa se cumple la siguiente relación sin carga a la salida:

$$V_{out}(DC) \simeq \sqrt{2}V_{in}(AC) \quad (A.1)$$

Esta expresión sitúa el nivel de salida en $24\sqrt{2} = 33.94VDC$

Con el objeto de obtener un voltaje de salida del convertidor en el rango de alimentación de la *PCM-3910* y lo más estable posible se decidió añadir un regulador, cuya función es eliminar cualquier rizado que pudiera quedar tras la rectificación. En este caso, por conveniencia se escogió uno de la serie 78XX, en concreto el 7818, con salida de 18VDC. Para otros niveles de voltaje se encuentran disponibles el 7805 (5VDC), 7808 (8VDC), 7809 (9VDC), 7812 (12VDC) y 7824 (24VDC). Estas salidas son las mínimas que entrega el componente, pudiéndose ajustar mediante resistencias adecuadas a cualquier nivel de voltaje, con la desventaja de que la diferencia de tensión entre entrada y salida será disipada en forma de calor en la carcasa del regulador.

En este último aspecto, fue necesario conocer la potencia consumida por el *PCM-3365*. En su *datasheet* se puede encontrar que en el caso más desfavorable es de alrededor de 7W, que se encuentra dentro de las posibilidades del regulador si se instala un disipador adecuado.

A.2 Materiales

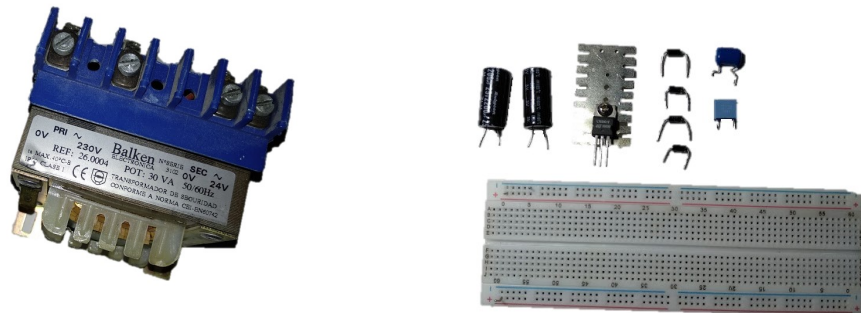
Según lo dispuesto anteriormente, se recogen a continuación los materiales necesarios para la fabricación de la fuente:

Transformador Potencia de 30VA y salida de 24 VAC

Diodos rectificadores 4 X 1N4001

Condensadores 2 X $2200\mu F$ @ 25V; 1 X 220nF; 1 X $0.1\mu F$

Regulador 1 X L7818C y disipador



(a) Transformador AC/AC .

(b) Componentes empleados .

Figura A.2 Materiales de la fuente.

A.3 Esquema

Por último se presenta el circuito eléctrico empleado

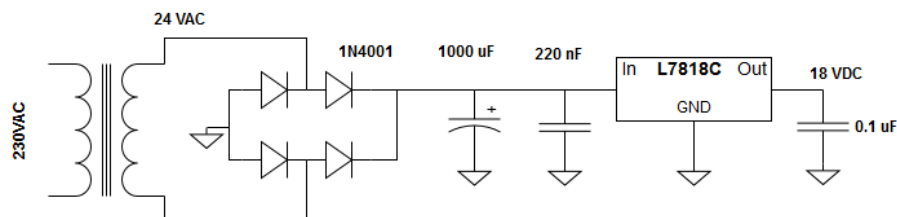


Figura A.3 Circuito de la fuente.

Apéndice B

Histogramas y medición de tiempos con MATLAB

En este anexo se desarrolla la manera de obtener los datos de salida de *cyclictest* ejecutado en el sistema embarcado, transferirlos a un PC y graficarlos en formato histograma. Asimismo, según se expone en la sección 5.4, se facilita el código para determinar los tiempos entre recepción de mensajes CAN.

B.1 Histogramas

B.1.1 Parámetros necesarios para *Cyclictest*

En primer lugar, para escribir los resultados de *cyclictest* en un archivo de texto es necesario emplear la sintaxis descrita en el código B.1

Código B.1 salida para *cyclictest*.

```
1 root@rtlinux:~# cd /usr/src/TFM/rt-tests-1.3
2 root@rtlinux:~# hackbench -s1000 -l1000000
3 root@rtlinux:~# cyclictest -t -p99 -i1000 -h50 --histfile=/usr/src/TFM/rt-tests
  -1.3/histograma.txt
```

El anterior código ejecuta en primer lugar el *script* de carga *hackbench* con los parámetros descritos en 3.4.2. Tras enviarlo al *background*, el programa *cyclictest* expide los datos al fichero *histograma.txt* mediante los siguientes comandos adicionales:

-h50 proporciona el histograma hasta detectarse una latencia máxima de 50 μ s

--histfile=\dirección\del \fichero ruta del archivo de salida.

Una vez terminada la prueba, es necesario transferir el fichero *histograma.txt* al PC del laboratorio mediante un USB por ejemplo.

B.1.2 Programa MATLAB *plot_histograma.m*

El programa empleado para dibujar el histograma fue MATLAB. A continuación se adjunta el código utilizado.

Código B.2 Código del programa para dibujar histogramas.

```
1 clear; clc; close all;
2
3 %% Lectura de datos
4
```

```

5 fileID1=fopen('histograma.txt','r'); % Abrir el archivo
6
7 formatSpec= '%f %f %f %f %f'; %Formato de columnas del archivo de texto
8
9 sizeA = [5 Inf];
10 A = fscanf(fileID1,formatSpec,sizeA);
11 fclose(fileID1);
12 A=A';
13 k=1;
14
15 %% Gráfico
16
17 figure(1);
18
19 bar(1:100,A(:,2),0.5); hold on;
20 bar(1:100,A(:,3),0.5); hold on;
21 bar(1:100,A(:,4),0.5); hold on;
22 bar(1:100,A(:,5),0.5);
23
24 %% Formato del gráfico
25
26 axis([0 20 0 6e7]);
27 legend('CPU1', 'CPU2', 'CPU3', 'CPU4');
28 title('Histogramas de \textit{cyclicttest}','Interpreter','latex');
29 ylabel('Número de bucles');
30 xlabel('Latencia $(\mu s)$','Interpreter','latex');
31
32 grid;

```

B.2 Determinación de tiempos entre mensajes CAN

Seguidamente se adjunta el código empleado en la sección 5.4

Código B.3 Código del programa para graficar el tiempo entre mensajes CAN.

```

1 clear all; close all; clc;
2 A = fscanf(fileID1,formatSpec,sizeA);
3
4 fclose(fileID1);
5
6 figure(1);
7 plot(1:length(A),A*1e3);
8 title('Tiempo de ejecución vs número de bucle');
9 xlabel('Bucle');
10 ylabel('Tiempo de ejecución (ms)');
11 grid on;
12
13 figure(2);
14 histogram(A,0.01996:1e-6:0.020015);
15 title('Histograma de los tiempos de ejecución');
16 xlabel('Periodo (s)');
17 ylabel('Número de bucles');
18 grid on;

```


Apéndice C

Códigos del demostrador de concepto

En este anexo se presentan los códigos empleados en la construcción del demostrador de concepto presentado en la sección 5.3.3, referente al análisis del código del controlador propuesto.

C.1 Unidad de Control Electrónico (*PCM3365* y *PCM-3680I*)

Código C.1 Código de ejemplo del controlador.

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 #include <sched.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <net/if.h>
8 #include <sys/types.h>
9 #include <sys/socket.h>
10 #include <sys/ioctl.h>
11 #include <linux/can.h>
12 #include <linux/can/raw.h>
13 #include <linux/can/error.h>
14 #include <fcntl.h>
15 #include <sys/time.h>
16 #include <sys/timerfd.h>
17
18 int periodo = 20000; // Periodo en us
19
20 pthread_t h1; //Se crea el thread id del hilo 1
21 pthread_t h2; //Se crea el thread id del hilo 2
22
23 struct periodic_info {
24 int timer_fd;
25 unsigned long long wakeups_missed;
26 };
27 int j;
28
29 static int make_periodic(unsigned int period, struct periodic_info *info) //
    INICIO FUNCIÓN MAKE_PERIODIC()
30 {
31 int ret;
32 unsigned int ns;
```

```

33 unsigned int sec;
34 int fd;
35 struct itimerspec itval;
36 //Crear timer
37 fd = timerfd_create(CLOCK_MONOTONIC, 0);
38 info->wakeups_missed = 0;
39 info->timer_fd = fd;
40 if (fd == -1)
41 return fd;
42
43 //Hacer el timer periódico
44 sec = period / 1000000;
45 ns = (period - (sec * 1000000)) * 1000;
46 itval.it_interval.tv_sec = sec;
47 itval.it_interval.tv_nsec = ns;
48 itval.it_value.tv_sec = sec;
49 itval.it_value.tv_nsec = ns;
50 ret = timerfd_settime(fd, 0, &itval, NULL);
51 return ret;
52 }
53
54 static void wait_period(struct periodic_info *info) //INICIO FUNCIÓN
    WAIT_PERIOD
55 {
56 unsigned long long missed;
57 int ret;
58 //Esperar al siguiente evento del timer. Si se ha excedido alguno, el número se
    escribe en "missed"
59 ret = read(info->timer_fd, &missed, sizeof(missed));
60
61 // read se bloquea si no ha pasado del tiempo del timer
62 if (ret == -1) {
63 perror("read timer");
64 return;
65 }
66
67 info->wakeups_missed += missed;
68 if(missed>1){
69 printf("Missed = %lld\n",missed);
70 }
71 }
72
73
74 static void* fun_hilo1(void *arg){ /*INICIO FUNCIÓN DEL HILO 1.*/
75
76 struct periodic_info info;
77
78 //Escribir aquí el programa de la ECU
79 int s,i;
80 int nbytes;
81 int nbytes2;
82 struct sockaddr_can addr;
83 struct can_frame frame;
84 struct can_frame frame2;
85 struct ifreq ifr;
86 char *ifname = "can0";
87

```

```

88 int setflag,getflag,ret =0;
89 if((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0){
90 perror("Error while opening socket");
91 return (void*)-1;
92 }
93
94 strcpy(ifr.ifr_name, ifname);
95 ioctl(s, SIOCGIFINDEX, &ifr);
96
97
98 addr.can_family = AF_CAN;
99 addr.can_ifindex = ifr.ifr_ifindex;
100
101 if(bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0){
102 perror("Error in socket bind");
103 return (void*)-2;
104 }
105
106
107 setflag = setflag|O_NONBLOCK;
108 ret = fcntl(s,F_SETFL,setflag);
109 getflag = fcntl(s,F_GETFL,0);
110
111 can_err_mask_t err_mask = CAN_ERR_TX_TIMEOUT |CAN_ERR_BUSOFF;
112 ret = setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, &err_mask, sizeof(err_mask
    ));
113
114 // printf("Interfaz ECU %s", ifname);
115
116 make_periodic(periodo, &info);
117
118 while(1){ // BUCLE PRINCIPAL
119 // printf("\n He entrado en el bucle while de la lectura");
120 nbytes = read(s, &frame, sizeof(frame));
121 // printf("\n nbytes = %d",nbytes);
122 if(frame.can_id == 1){
123 // printf("leido 1");
124 if (nbytes > 0){
125 if (frame.can_id & CAN_ERR_FLAG)
126 printf("error frame\n");
127 else
128 {
129 //printf("\n ID=%d \n ",frame.can_id);
130 //for (i =0;i<frame.can_dlc;i++)
131 // {
132 //     printf("data[%d]=%d",i,frame.data[i]);
133 //     }
134 }
135 }
136 //ESCRITURA DE LA SENAL DE CONTROL
137 // printf("\n empiezo a escribir señal de control");
138
139 frame2.can_id = 2;
140 frame2.can_dlc = 5;
141
142 if(frame.data[0]<=25){
143 frame2.data[0] = 1;

```

```

144 frame2.data[1] = 0;
145 frame2.data[2] = 0;
146 frame2.data[3] = 0;
147 }
148 else if(frame.data[0]>25 && frame.data[0]<=50){
149 frame2.data[0] = 0;
150 frame2.data[1] = 1;
151 frame2.data[2] = 0;
152 frame2.data[3] = 0;
153 }
154 else if(frame.data[0]>50 && frame.data[0]<=75){
155 frame2.data[0] = 0;
156 frame2.data[1] = 0;
157 frame2.data[2] = 1;
158 frame2.data[3] = 0;
159 }
160 else{
161 frame2.data[0] = 0;
162 frame2.data[1] = 0;
163 frame2.data[2] = 0;
164 frame2.data[3] = 1;
165 }
166 frame2.data[4] = frame.data[0]; //se lee directamente la posición del
167 //potenciómetro de los mensajes enviados por la mbed y se le devuelven
168 //sin modificar a través del bus CAN
169 nbytes2 = write(s, &frame2, sizeof(struct can_frame));
170 }
171 usleep(100); //Esta función es un stopping point para pthread_cancel.
172
173 wait_period(&info); //Función para esperar el periodo de ejecución
174
175 }/*FIN DEL BUCLE PRINCIPAL*/
176 pthread_exit(NULL);
177 }/*FIN DE LA FUNCIÓN DEL HILO 1*/
178
179
180
181 void* fun_hilo2(){
182 //Esta función interrumpe a la función 1. Es una parada de emergencia
183 //printf("Modo de parada activado \n");
184 int j;
185 int flag = 0;
186 while(flag == 0){
187 scanf(" %d",&j);
188 if (j==(int)5){
189 flag = 1;
190 printf("\n !!!!!!!!!!!!! PARADA DE EMERGENCIA !!!!!!!!!!!!!\n");
191 pthread_cancel(h1); //cierra el hilo principal
192 pthread_exit(NULL); //terminar hilo
193 } else{
194 printf("Orden no reconocida");
195 }
196 }
197
198
199
200 }

```

```
201
202 // NOTA: Typcasting = pasar de un tipo a otro;
203 // La sintaxis es (newvartype)name
204
205
206
207 int main(){ //Función principal
208 /*INICIO DE LA FUNCIÓN PRINCIPAL*/
209
210 int policy1;
211 int policy2;
212
213 struct sched_param params1;
214 struct sched_param params2;
215
216 params1.sched_priority = 30;
217 params2.sched_priority = 40;
218
219
220 pthread_create(&h1, NULL, fun_hilo1, NULL); //funcion de llamada del hilo. Se
    necesitan
221 //cuatro argumentos: La direccion del hilo, los atributos del mismo, la
    direccion de la funcion del hilo y los
222 // parametros de entrada.
223
224 if(pthread_setschedparam(h1, SCHED_FIFO, &params1)!=0){
225 printf("SIN ÉXITO AL CAMBIAR LA POLICY 1 /n");
226 }
227
228 int ret1 = pthread_getschedparam(h1, &policy1, &params1);
229
230
231
232 pthread_create(&h2, NULL, fun_hilo2, NULL); //Creamos el hilo de parada de
    emergencia.
233
234 if(pthread_setschedparam(h2, SCHED_FIFO, &params2)!=0){
235 printf("SIN ÉXITO AL CAMBIAR LA POLICY 2 /n");
236 }
237
238 int ret2 = pthread_getschedparam(h2, &policy2, &params2);
239 /**/
240
241
242
243 int comp1 = pthread_join(h1,NULL); //Funcion que indica si el hilo 1 ha
    terminado y devuelve si se quiere el estado de cierre.
244 int comp2 = pthread_join(h2,NULL); //Funcion que indica si el hilo 1 ha
    terminado y devuelve si se quiere el estado de cierre.
245
246 //COMPROBACIONES PARA EL HILO 1
247 if(comp1==0){
248 printf("\n\n\n---- COMPROBACIONES----\n\n");
249 printf("\n hilo 1 terminado \n");
250 }
251 else{
252 printf("Fallo en terminar el hilo 1");
```

```

253 }
254
255 if (policy1!=SCHED_FIFO){
256 printf("NO FUNCIONA LA POLICY 1\n");
257 printf("%d \n",(int)policy1);
258 }
259 else{
260 printf("POLICY 1 SCHED_FIFO \n");
261 }
262 printf("la prioridad del hilo 1 es %d \n",params1.sched_priority);
263
264 //COMPROBACIONES PARA EL HILO 2
265
266
267 if(comp2==0){
268 printf("hilo 2 terminado \n");
269 }
270 else{
271 printf("Fallo en terminar el hilo 2");
272 }
273
274 if (policy1!=SCHED_FIFO){
275 printf("NO FUNCIONA LA POLICY 2\n");
276 printf("%d \n",(int)policy1);
277 }
278 else{
279 printf("POLICY 2 SCHED_FIFO \n");
280 }
281 printf("la prioridad del hilo 2 es %d \n",params2.sched_priority);
282 /**/
283
284 }

```

C.2 Código de la tarjeta de adquisición de datos y actuadora (Mbed LPC1768)

Código C.2 Código de ejemplo para la tarjeta *LPC-1768*.

```

1 #include "mbed.h"
2
3 //Este programa toma la posición de un potenciómetro, la manda a la ECU a través
  //del bus CAN y escribe el nivel que ésta devuelve en los LEDs indicadores
  //de la mbed, divididos en cuatro regiones. Además, permite el control de un
  //servomotor mediante PWM.
4
5 // Inicialización de los LEDs
6 DigitalOut led_low(LED1); //LED que se enciende en el primer cuarto del potenci
  //ómetro
7 DigitalOut led_mid_low(LED2); //Segundo cuarto del potenciómetro
8 DigitalOut led_mid_high(LED3); //Tercer cuarto del potenciómetro
9 DigitalOut led_high(LED4); //Último cuarto del potenciómetro
10
11 //Inicilización de la señal de control del servo
12 PwmOut servo(p21);
13

```

```
14 //Inicialización de la entrada analógica para el potenciómetro
15 AnalogIn pot(p20);
16 char pot_c;
17
18 //Inicialización de la interfaz CAN
19 CAN can1(p30, p29);
20
21 //Inicialización de variable para guardar mensaje que se lee
22 CANMessage leido;
23
24
25 int main() {
26
27 servo.period(0.020); //Frecuencia de la señal PWM para el servo (20ms)
28 while(1) {
29
30 can1.frequency(250000); //Frecuencia del bus CAN
31
32 //Se crea un mensaje CAN con identificador 1 para escribir en el bus la
33 //posición del potenciómetro.
34
35 pot_c = floor(pot*100);
36
37 if(can1.write(CANMessage(1, &pot_c, 1))){
38 printf("\n");
39 t.start();
40 }
41
42 //Ahora se lee el bus para ejecutar la orden de la ECU, sólo se necesita
43 //leer los mensajes con identificador 2
44
45 can1.read(leido);
46
47 if (leido.id == 2){
48 //printf("He leído un mensaje con id=2");
49 t.stop();
50 printf("\n %f", (float)t);
51
52 led_low = leido.data[0];
53 led_mid_low = leido.data[1];
54 led_mid_high = leido.data[2];
55 led_high = leido.data[3];
56
57 float pw = 0.002*leido.data[4]/100+0.0003;
58 //printf("%f", pw);
59 servo.pulsewidth(pw);
60 }
61 }
62 }
```

C.3 Medición del tiempo entre paquetes recibidos

Código C.3 Código de ejemplo para medir tiempos entre recepción mediante la tarjeta *LPC-1768*.

```
1 #include "mbed.h"
2 #include <fstream>
3 #include <iostream>
4
5 using namespace std;
6
7 volatile float tiempo;
8
9 volatile bool imprime = false;
10
11 //Inicialización de la interfaz CAN
12 CAN can1(p30, p29);
13
14 //Inicialización de variable para guardar mensaje que se lee
15 CANMessage leido;
16
17 //Inicialización del timer
18 Timer t;
19
20 //Bandera de salida del bucle principal
21 int flag =0;
22
23 //Función de la interrupción
24 void inter(){
25
26 can1.read(leido);
27 if(leido.id==2){
28 flag=1;
29 }
30
31 t.stop();
32
33 tiempo = t.read();
34 imprime = true;
35 t.reset();
36 t.start();
37
38 }
39
40
41 //Inicio de la función principal
42 int main() {
43
44 cout<<"funcion principal"<<endl;
45 can1.frequency(250000);
46 can1.attach(&inter);
47
48 t.start();
49
50 while(flag==0){
51
```



```

52 if (imprime){
53 imprime=false;
54 cout << tiempo << endl;
55 }
56 }
57
58 cout << "fuera del bucle" <<endl;
59 cout<<"programa terminado"<<endl;
60 }

```

Código C.4 Código de ejemplo para enviar mensajes CAN de manera periódica mediante el montaje *PCM-3365* y *PCM-3680I*.

```

1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4  #include <sched.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <net/if.h>
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <sys/ioctl.h>
12 #include <linux/can.h>
13 #include <linux/can/raw.h>
14 #include <linux/can/error.h>
15 #include <fcntl.h>
16 #include <sys/time.h>
17 #include <sys/timerfd.h>
18
19 int periodo = 20000; //periodo en us
20 int bucles = 10000;
21 struct periodic_info {
22 int timer_fd;
23 unsigned long long n_perdidas;
24 };
25
26 static int make_periodic(unsigned int period, struct periodic_info *info)
27 {
28 int ret;
29 unsigned int ns;
30 unsigned int sec;
31 int fd;
32 struct itimerspec itval;
33
34 //Crear el timer
35 fd = timerfd_create(CLOCK_MONOTONIC, 0);
36 info->n_perdidas = 0;
37 info->timer_fd = fd;
38 if (fd == -1)
39 return fd;
40
41 //Hacer el timer periódico
42 sec = period / 1000000;
43 ns = (period - (sec * 1000000)) * 1000;

```

```
44 itval.it_interval.tv_sec = sec;
45 itval.it_interval.tv_nsec = ns;
46 itval.it_value.tv_sec = sec;
47 itval.it_value.tv_nsec = ns;
48 ret = timerfd_settime(fd, 0, &itval, NULL);
49 return ret;
50 }
51
52 static void wait_period(struct periodic_info *info)
53 {
54 unsigned long long perdidas;
55 int ret;
56
57 ret = read(info->timer_fd, &perdidas, sizeof(missed));
58 // read se para si no ha pasado del tiempo del timer
59 }
60
61 info->n_perdidas += perdidas;
62 if(perdidas>1){
63 printf("Perdidas = %lld\n",perdidas);
64 }
65 }
66
67 static int thread_1_count = 0;
68
69 static void *thread_1(void *arg) // FUNCIÓN DEL HILO
70 {
71 struct periodic_info info;
72
73 printf("Comienzo del hilo 1 \n");
74
75 //Comprobaciones para bus CAN
76
77 int s,i;
78 int nbytes;
79 struct sockaddr_can addr;
80 struct can_frame frame;
81 struct ifreq ifr;
82 char *ifname = "can0";
83
84 int setflag,getflag,ret =0;
85 if((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0){
86 perror("Error while opening socket");
87 return (void*)-1;
88 }
89
90 strcpy(ifr.ifr_name, ifname);
91 ioctl(s, SIOCGIFINDEX, &ifr);
92
93
94 addr.can_family = AF_CAN;
95 addr.can_ifindex = ifr.ifr_ifindex;
96
97 if(bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0){
98 perror("Error in socket bind");
99 return (void*)-2;
100 }
```

```

101
102
103 setflag = setflag|O_NONBLOCK;
104 ret = fcntl(s,F_SETFL,setflag);
105 getflag = fcntl(s,F_GETFL,0);
106
107 can_err_mask_t err_mask = CAN_ERR_TX_TIMEOUT |CAN_ERR_BUSOFF;
108 ret = setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, &err_mask, sizeof(err_mask
    ));
109
110 make_periodic(periodo, &info);
111
112 // Fin de comprobaciones para bus CAN
113
114 while (thread_1_count<bucles) { //Función principal del hilo
115
116 frame.can_id = 1;
117 frame.can_dlc = 1;
118
119 frame.data[0] = 1;
120
121 nbytes = write(s, &frame, sizeof(struct can_frame));
122
123 printf("%d\n",thread_1_count);
124 thread_1_count++;
125
126 wait_period(&info);
127 }
128 frame.can_id = 2; //Al final del bucle mandamos un mensaje con id=2 para parar
    el código
129 // de la mbed LPC-1768
130
131 frame.can_dlc = 1;
132
133 frame.data[0] = 1;
134
135 nbytes = write(s, &frame, sizeof(struct can_frame));
136
137 return NULL;
138 }
139
140
141 int main() //FUNCIÓN PRINCIPAL
142 {
143 printf("Comprobación de la peridicidad de pthreads\n");
144
145 pthread_t t_1;
146
147 int policy;
148 struct sched_param params;
149
150 params.sched_priority = 80;
151
152
153 pthread_create(&t_1, NULL, thread_1, NULL);
154
155 if(pthread_setschedparam(t_1, SCHED_FIFO, &params)!=0){

```

```
156 printf("SIN ÉXITO AL CAMBIAR LA POLICY /n");
157 }
158
159
160 int ret = pthread_getschedparam(t_1, &policy, &params);
161
162 sleep(10);
163
164 int comp = pthread_join(t_1, NULL);
165
166
167 //COMPROBACIONES PARA EL HILO
168 if(comp==0){
169 printf("\n\n\n---- COMPROBACIONES----\n\n");
170 printf("\n hilo terminado \n");
171 }
172 else{
173 printf("Fallo en terminar el hilo");
174 }
175
176 if (policy!=SCHED_FIFO){
177 printf("NO FUNCIONA LA POLICY\n");
178 printf("%d \n", (int)policy);
179 }
180 else{
181 printf("POLICY SCHED_FIFO \n");
182 }
183 printf("la prioridad del hilo es %d \n", params.sched_priority);
184
185
186 return 0;
187 }
```

Índice de Figuras

1.1.	Vehículo FOX	1
1.2.	Entidades promotoras del proyecto CONFIGURA	1
1.3.	Esquema general de las conexiones del vehículo	3
1.4.	Equipo de pruebas empleado en [1]	3
1.5.	Resultados de latencia	4
2.1.	<i>SBC PCM-3365</i>	6
2.2.	Fuente <i>PCM-3910</i>	6
2.3.	Bloques funcionales	7
2.4.	Espaciadores	8
2.5.	Vista inferior de la placa donde se pueden apreciar las ranuras para la tarjeta SATA y para el módulo de memoria RAM	9
2.6.	Conectores	9
2.7.	Conexión de la fuente ATX	10
2.8.	Conexiones restantes	10
3.1.	Logotipo de <i>Slackware</i>	11
3.2.	Ventanas de <i>UNetbootin</i> y <i>Rufus</i>	12
3.3.	<i>Pinout</i> de CN10 [3]	13
3.4.	Pantalla de inicio	13
3.5.	Ventanas de instalación	14
3.6.	Particiones necesarias	14
3.7.	Opciones de instalación	15
3.8.	Partición objetivo	15
3.9.	Opción USB	15
3.10.	ventana de <i>menuconfig</i>	17
3.11.	Histograma de latencias de los núcleos	19
3.12.	Segunda prueba de rendimiento	20
4.1.	Modelo OSI del protocolo CAN [8]	22
4.2.	Modelo de hardware CAN	22
4.3.	pinout del MCP2551	22
4.4.	Niveles lógicos CAN	23
4.5.	Voltaje diferencial	24
4.6.	Módulo CAN <i>PCM-3680</i>	25
4.7.	Soporte de las versiones del <i>Kernel</i>	25
4.8.	Carga de <i>drivers</i>	26
4.9.	lista de enlaces de datos	27
4.10.	CAN activado	27
4.11.	Resultado de la comunicación CAN	29
4.12.	Detalles del microcontrolador	30

4.13.	Apariencia del dispositivo	30
4.14.	Ventanas de instalación	31
4.15.	Ventana de opciones	31
4.16.	Esquema de la prueba [13]	32
4.17.	Protoboard con las conexiones necesarias	33
4.18.	Visualización a través de terminal	34
4.19.	Mensaje de prueba en el terminal	35
4.20.	Mejoras de <i>Putty</i>	35
4.21.	Salida del programa de intercomunicación	36
4.22.	Diagrama de conexión entre placas	37
4.23.	<i>Pinout</i> del conector DB9	37
4.24.	Salidas por pantalla de la primera prueba	40
5.1.	Esquema de procesos e hilos	42
5.2.	Servomotor <i>Futaba S3003</i>	45
5.3.	Señal PWM con sus características principales	45
5.4.	Montaje de prueba del servo	46
5.5.	Diagrama de flujo de la función del controlador del vehículo	47
5.6.	Diagrama de flujo de la función del hilo principal	48
5.7.	Diagrama de flujo de la función principal	48
5.8.	Diagrama de flujo del código en la <i>Mbed LPC1768</i>	51
5.9.	Histograma de latencias del montaje completo	52
5.10.	Diagrama de flujo de los dos dispositivos durante la prueba	53
5.11.	Opciones para crear un <i>log file</i>	55
5.12.	Histograma de tiempos entre mensajes	56
5.13.	Distribución parcial de tiempos de ejecución	57
6.1.	Esquemas de pruebas propuestos	60
A.1.	Esquema con convertidor	61
A.2.	Materiales de la fuente	62
A.3.	Circuito de la fuente	62

Índice de Tablas

2.1.	Características de la tarjeta PCM3365	5
2.2.	Características de la tarjeta PCM3910	6
4.1.	Tabla de <i>pinout</i>	23
4.2.	Características de la tarjeta <i>PCM-3680I</i>	24
5.1.	<i>Características del servomotor Futaba S3003</i> [19]	45

Índice de Códigos

3.1.	Montar dispositivo USB	16
3.2.	Copia de archivos	16
3.3.	Extracción del kernel y el parche RT	16
3.4.	Aplicar el parche	16
3.5.	Proceso de compilación	16
3.6.	Proceso de compilación (continuación)	17
3.7.	lilo.conf	18
3.8.	Comprobación	18
3.9.	Ejecución de las pruebas	19
4.1.	Extracción de <i>drivers</i>	25
4.2.	Carga automática de <i>drivers</i>	26
4.3.	Inclusión de <i>drivers</i> CAN	26
4.4.	Modificación en el fichero rc.local	27
4.5.	Programa readCAN	28
4.6.	Programa readCAN	28
4.7.	Programa de prueba	34
4.8.	Programa de prueba para comunicación CAN	35
4.9.	Programa para escribir en el bus CAN	37
4.10.	Programa de ejemplo <i>readCAN</i>	38
5.1.	Adjuntando la librería <i>pthread</i>	44
5.2.	Programa de prueba para el servomotor	46
5.3.	Comandos <i>shell</i> para ejecutar el programa del controlador en segundo plano	52
B.1.	salida para <i>cyclictest</i>	63
B.2.	Código del programa para dibujar histogramas	63
B.3.	Código del programa para graficar el tiempo entre mensajes CAN	64
C.1.	Código de ejemplo del controlador	65
C.2.	Código de ejemplo para la tarjeta <i>LPC-1768</i>	70
C.3.	Código de ejemplo para medir tiempos entre recepción mediante la tarjeta <i>LPC-1768</i>	72
C.4.	Código de ejemplo para enviar mensajes CAN de manera periódica mediante el montaje <i>PCM-3365</i> y <i>PCM-3680I</i>	73

Bibliografía

- [1] Pablo Marín Cortés. Estudio Preliminar del Diseño de una Nueva Unidad de Control Electrónico del Motor del Vehículo FOX, 2017.
- [2] Página web del proyecto CONFIGURA dentro de la web del Centro Nacional del Hidrógeno. <https://www.cnh2.es/cnh2/configural/>, 2019.
- [3] Advantech. PCM-3365 Datasheet, 2017.
- [4] Advantech. PCM-3910 DC to DC power supply PC / 104 + Module Startup Manual, 2005.
- [5] Página principal de Rufus. <https://rufus.akeo.ie/?locale>.
- [6] Página principal de Unetbootin. <https://unetbootin.github.io/>.
- [7] Kernel Documentation Projects. https://docs.slackware.com/slackbook:linux_kernel.
- [8] Pat Richards. A CAN Physical Layer Discussion. <https://os.mbed.com/handbook/SerialPC>, 2002.
- [9] Microchip Technologies inc. MCP 2551 datasheet . <http://ww1.microchip.com/downloads/en/devicedoc/21667e.pdf>, Consultado en 2019.
- [10] Advantech. PCM-3680I datasheet , Consultado en 2019.
- [11] Advantech. Documentación de los drivers de la placa PCM-3680I. https://support.advantech.com/support/DownloadSRDetail_New.aspx?SR_ID=GF-GRSC&Doc_Source=Download, 2019.
- [12] Mbed ARM. Descripción del pinout de la tarjeta Mbed LPC-1768. <https://os.mbed.com/platforms/mbed-LPC1768/>, 2019.
- [13] David Smart. CAN-Getting started. <https://os.mbed.com/users/WiredHome/notebook/can---getting-started/>, 2018.
- [14] ARM Mbed. Mbed Handbook. SerialPC. <https://os.mbed.com/handbook/SerialPC>.
- [15] ARM Mbed. Mbed Handbook. Terminal communication. <https://os.mbed.com/handbook/Terminals>.
- [16] Rob Toulson and Tim Wilmshurst. Fast and effective embedded systems design, 2012.
- [17] Dave McCracken. IBM Linux Technology Center. *Proceedings of the Ottawa Linux Symposium: POSIX Threads and the Linux Kernel. pág 330* , 2002. Consultado en 2019.
- [18] Página del manual de Linux relativa a pthreads. <http://man7.org/linux/man-pages/man7/pthreads.7.html>, 2019.
- [19] Futaba corp. Datasheet del servomotor Futaba S3003 . <https://www.rc.futaba.co.jp/english/servo/products/s3003.html>, Consultado en 2019.
- [20] Juan Gómez González. Andrés Prieto-Moreno Torres. Documentación técnica y método de conexión del servomotor Futaba S3003. <http://www.learobotics.com/proyectos/cuadernos/ct3/ct3.html>, 2019.

- [21] Developing the pulse width modulation tool (pwmt) for two timer mechanism technique in microcontrollers - scientific figure on researchgate. https://www.researchgate.net/figure/PWM-signal-with-its-two-basic-time-periods_fig4_271437313.
- [22] *Linux man pages*. Documentación de la función `timerfd_create`. man7.org/linux/man-pages/man2/timerfd_create.2.html, Consultado en 2019.
- [23] *Mbed ARM*. Documentación la API del bus CAN. <https://os.mbed.com/docs/mbed-os/v5.14/apis/can.html>, Consultado en 2019.
- [24] *Mbed ARM*. Documentación las funciones miembro de la clase CAN: `can.attach()`. https://os.mbed.com/docs/mbed-os/v5.14/mbed-os-api-doxy/classmbed_1_1_c_a_n.html#a554dccfc95f8e864de7be65a4a793e98, Consultado en 2019.
- [25] Jan Jongboom. *Simplify your code with mbed-events*. <https://os.mbed.com/blog/entry/Simplify-your-code-with-mbed-events/>, Consultado en 2019.
- [26] *Mbed ARM. Foros de preguntas*. Explicación relativa a los problemas en el uso de funciones como `printf()` con interrupciones ISR. <https://os.mbed.com/questions/68656/Add-interrupt-into-queue/>, Consultado en 2019.
- [27] Alfonso Valenzuela Romero. Apuntes de la asignatura *Tráfico Aéreo Avanzado* del Máster Universitario en Ingeniería Aeronáutica de la Universidad de Sevilla. Tema 6. Validación de Conceptos Operacionales en ATM. Sección 6.4.2. Pág. 97. , Consultado en 2019.