Universidad Autónoma de Coahuila

Facultad de Sistemas

Ingeniería en Electrónica y Comunicaciones

Implementación de una red can bus 2.0b utilizando microcontroladores

Autor:
Alfredo Valdés Cárdenas
Director de tesis:
M.I. Francisco Gerardo Hernández Rivera

Índice

Índice de figuras

Índice de cuadros

Nombre del proyecto

Implementación de una red CAN bus 2.0B utilizando microcontroladores.

Introducción

En la región de Saltillo existe una gran cantidad de empresas dedicadas a la industria automotriz, por lo que la investigación y desarrollo de tecnologías aplicables a la electrónica automotriz representa una opción de proyecto de tesis bastante obvia.

El protocolo de comunicación serial CAN (Controller Area Network) es utilizado de manera amplia en los automóviles, para transmitir la información, por ejemplo, de la velocidad de auto, las revoluciones por minuto que da el motor, el control de las bolsas de aire, del aire acondicionado, de las luces, de las ventanas eléctricas y los seguros, de la presión de aire en las llantas, incluso proporciona información como cuantos grados se ha girado el volante, la cantidad de gasolina en el tanque y fallas en el motor, como las que se obtienen con un escáner OBD-II.

En la realización de este proyecto se aplican conocimientos adquiribles durante la duración de la carrera de Ingeniero en Electrónica y Comunicaciones como la utilización de microcontroladores, la implementación de protocolos de comunicaciones y el diseño de circuitos electrónicos, por lo cual es posible asumir que un aspirante a obtener el título de Ingeniero en Electrónica y Comunicaciones pueda realizar el proyecto que se detalla en este documento.

En el ámbito personal, este proyecto representa un reto interesante que se desarrolla sobre una de los ámbitos que más me interesan de la electrónica: Los microcontroladores. Trabajar sobre una plataforma nueva, en una arquitectura más avanzada que la que se llega a ver en las clases que se cubren en la carrera me permite reafirmar los conocimientos previamente adquiridos en el área, así como incrementar las técnicas de desarrollo de hardware y software que poseo. El proyecto también me ayuda a cimentar la orientación a las comunicaciones, una de las áreas con más demanda en México en la actualidad

Antecedentes

En Febrero de 1986, Robert Bosch GmbH presento el sistema de bus serial Controller Area Network (CAN) en el congreso de la Society of Automotive Engineers (SAE). Hoy en día, casi todos los vehículos de pasajeros que se fabrican en el mundo vienen equipados con al menos una red CAN. Utilizado también en otros tipos de vehículos, desde trenes a barcos, así como en controles industriales, CAN es uno de los protocolos de bus dominantes a nivel mundial - tal vez sea el mas usado a nivel mundial. A principios de la década de 1980, los ingenieros de Bosch estaban evaluando los sistemas de bus serial existentes para determinar su posible uso en automóviles. Dado que ninguno de los protocolos de red disponibles cumplía con los requisitos que buscaban los ingenieros, Uwe Kiencke comenzó a desarrollar un nuevo sistema de bus serial en 1983.

El nuevo protocolo solo pretendía añadir nuevas funcionalidades - la reducción de la cantidad de cableado y arneses requerida fue una consecuencia, mas no la razón detrás del desarrollo de CAN. Ingenieros de Mercedes-Benz se unieron al proyecto en los inicios de la fase de especificación del nuevo sistema, así como Intel, que buscaba ser el proveedor de semiconductores principal del proyecto. El Profesor Wolfhard Lawrenz de la Universidad de Ciencia Aplicada en Braunschweig-Wolfenbüttel, Alemania, quien había sido contratado como un consultor, le dio al nuevo protocolo de red el nombre 'Controller Area Network'. El Profesor Horst Wettstein de la Universidad de Karlsruhe aporto apoyo académico a la investigación.[8]

En la actualidad el protocolo CAN se encuentra en su tercera iteración, denominada CAN FD; esta tecnología se encuentra ya implementada en los automóviles mas modernos, sin embargo la adquisición de plataformas de desarrollo para el consumidor en general todavía es limitada, por lo que en este proyecto se utilizara la versión anterior y la que tiene mayor propagación a nivel mundial, el CAN 2.0B.

Marco Teórico

Introducción al bus CAN

Controller Area Network, (CAN, por sus siglas en ingles), es un protocolo de comunicación serial estandarizado a nivel internacional por ISO[1].

La industria automotriz ha podido observar en su tiempo de vida distintos sistemas de control electrónicos que han sido desarrollados teniendo en mente características como seguridad, confort, prevención de contaminación y bajo costo. Sin embargo, estos sistemas presentan una desventaja en que los tipos de datos de comunicación, un nivel de confiabilidad requerida, etc. . . , diferían entre cada sistema, y se debían configurar en múltiples líneas de bus que requerían una mayor cantidad de cableado.

Existe así pues una necesidad por reducir la cantidad de cables y poder transferir grandes cantidades de información atravesó de múltiples redes locales. Para satisfacer esta necesidad, BOSCH, una empresa alemana de equipo eléctrico, desarrollo CAN en 1986 como un protocolo de comunicación para automóviles. Después CAN se estandarizo en ISO 11898 e ISO 11519, estableciéndose como el protocolo estándar de redes vehiculares. Hoy en día, CAN es ampliamente aceptado por su gran desempeño y confiabilidad, y es utilizado en una gran variedad de industrias desde lo automotriz hasta la industria médica.

Características y prestaciones del bus CAN

- Económico y sencillo de implementar: El hecho de poder ahorrar en cableado y, sobre todo en el diseño de los dispositivos en el aspecto de comunicaciones, reduce el costo y la complejidad del diseño del circuito.
- Gestión de prioridades: Al momento de ocurrir una colisión de paquetes en el bus, se le transmitirá el mensaje con mayor prioridad.
- Gestión inteligente del bus: El bus CAN realiza una gestión de la transferencia de mensajes de bit inteligente de manera que, si en un momento dado de la transmisión, un mensaje resulta ser más prioritario que los demás, no se volverá a retransmitir desde el principio y continuará su transmisión.
- Mensajes o CAN frames: Los mensajes que se transmiten al bus CAN, se denominan "CAN frames" que contienen una serie de cabeceras, un identificador y los datos del mensaje. El mensaje en formato estándar puede variar de 44 a 108 bits y, en el formato extendido, varía entre 64 a 128 bits.
- Control de errores de mensajes: La especificación del protocolo CAN, define una de las cabeceras del "CAN frame" para el control de errores de los mensajes. Dicho control es posible gracias al Checksum CRC de 15 bits establecido en el CAN frame para verificar la integridad del mensaje.
- Retransmisión de mensajes: Si un nodo no ha podido transmitir su mensaje debido a que su mensaje es menos prioritario que los otros, o no ha podido debido a un error de transmisión, el mensaje se retransmitirá cuantas veces sea necesario.
- Extensible: Es posible ampliar el número de bits usados en el "CAN frame" de forma sencilla, para extender el uso del protocolo, configurando el nodo concreto.
- Orientado a mensajes: Debido a que la especificación del protocolo CAN no contempla cabeceras específicas, para identificar en el "CAN frame" ni el remitente ni el destinatario, está orientado a mensaje debido a las características de sus cabeceras. Así pues, estos mensajes se les atribuyen unos identificadores en particular en la red y los nodos CAN, según sus necesidades, aceptarán dicho mensaje o no.

[1] Organización Internacional de estandarización, en ingles International Organization for Standarization.

- Tolerancia a errores de nodos: Si un nodo alcanza el número máximo de errores de transmisión o, por algún motivo, resulta ser defectuoso, no perjudicará el resto de los componentes o nodos de la red CAN, ya que aislarán de la comunicación al nodo conflictivo.
- Ancho de banda regulable: Para cualquier nodo CAN, es posible regular la velocidad a la que transmite de forma sencilla. Estas velocidades de transmisión comprenden entre 125 kbps (baja velocidad tolerante a fallos) y 1 Mbps.
- Broadcasting: Los mensajes que un nodo envía a través de la red CAN, es recibido por todos los nodos que estén conectados a la misma.
- Modelo de comunicación multimaestro: Todos y cada uno de los nodos CAN, disponen de la posibilidad de transmitir y recibir mensajes a cualquiera de los nodos sin ninguna restricción.

Modelo de referencia OSI y el Bus CAN

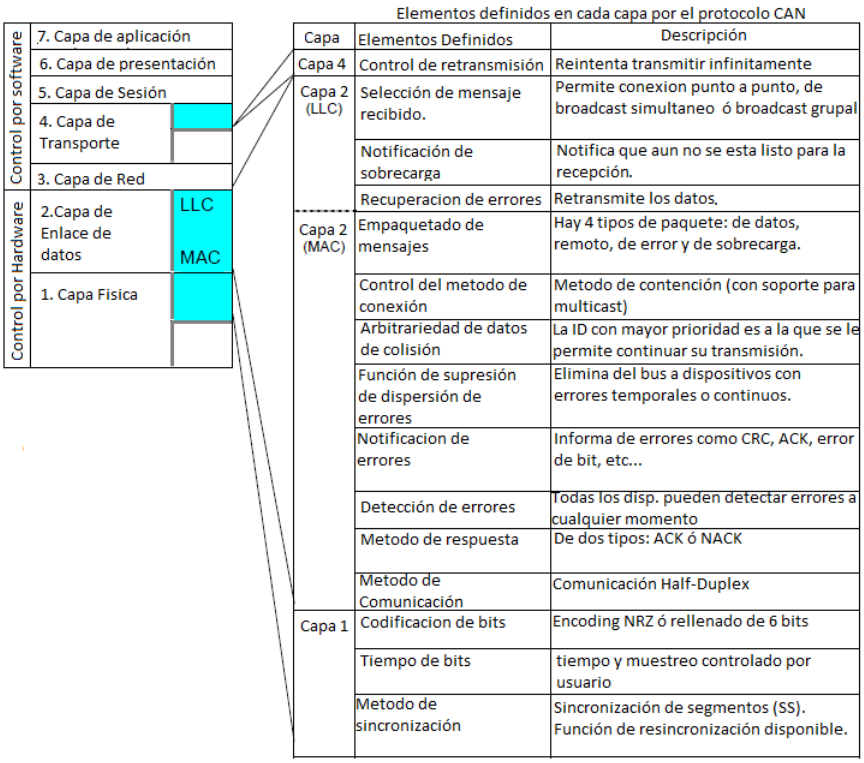
El protocolo CAN incluye las capas de transporte, enlace de datos y física del modelo de referencia básico OSI[2]. La capa de enlace de datos se divide en las subcapas MAC y LLC; La subcapa MAC constituye el núcleo del protocolo CAN.

La función de la capa de enlace de datos es unir las señales recibidas por la capa física en un mensaje estructurado que provee un proceso para el control de la transmisión de datos. De manera más específica, se trata de empaquetar los mensajes en un frame, Arbitrariedad de datos de colisión, un mensaje de acknowledgement (similar a un handshake) y detectar o notificar errores.
Estas funciones de la capa de enlace de datos se ejecutan normalmente mediante hardware en el controlador CAN.

Para la capa física, el protocolo define la manera en la cual las señales son transmitidas, el tiempo entre bits, la codificación de los bits y el proceso de sincronización. Sin embargo, esto no significa que los niveles de las señales, la velocidad de comunicación, los valores de muestreo, las características eléctricas del bus y el factor de forma del conector se definan de manera específica por el protocolo CAN; Todas estas deben de ser seleccionadas para cada dispositivo por el usuario.

En la especificación CAN de BOSCH, no existen definiciones con respecto a las características eléctricas de transceivers y bus. No obstante, en el estándar ISO para el protocolo CAN (ISO11898 e ISO 11519-2) las características físicas y eléctricas de los transceivers y el bus si están definidas.

[2]Modelo de interconexión de sistemas abiertos (ISO/IEC 7498-1), más conocido como "modelo OSI" (en inglés, Open System Interconnection)



Elementos definidos en cada capa por el protocolo CAN

Capa	Elementos Definidos	Descripción
Capa 4	Control de retransmisión	Reintenta transmitir infinitamente
Capa 2 (LLC)	Selección de mensaje recibido.	Permite conexion punto a punto, de broadcast simultaneo ó broadcast grupal
	Notificación de sobrecarga	Notifica que aun no se esta listo para la recepción.
	Recuperacion de errores	Retransmite los datos.
Capa 2 (MAC)	Empaquetado de mensajes	Hay 4 tipos de paquete: de datos, remoto, de error y de sobrecarga.
	Control del metodo de conexión	Metodo de contención (con soporte para multicast)
	Arbitrariedad de datos de colisión	La ID con mayor prioridad es a la que se le permite continuar su transmisión.
	Función de supresión de dispersión de errores	Elimina del bus a dispositivos con errores temporales o continuos.
	Notificacion de errores	Informa de errores como CRC, ACK, error de bit, etc...
	Detección de errores	Todas los disp. pueden detectar errores a cualquier momento
	Metodo de respuesta	De dos tipos: ACK ó NACK
	Metodo de Comunicación	Comunicación Half-Duplex
Capa 1	Codificacion de bits	Encoding NRZ ó rellenado de 6 bits
	Tiempo de bits	tiempo y muestreo controlado por usuario
	Metodo de sincronización	Sincronización de segmentos (SS). Función de resincronización disponible.

Figura 1: Relación entre el modelo OSI y el protocolo CAN (Traducción Propia).[1]

La Capa Física

La especificación del protocolo CAN está basado en los dos estándares de transmisión ISO 11898 (transmisión de alta velocidad) e ISO 11519 (transmisión de baja velocidad tolerante a fallos).

Estándar 11519:

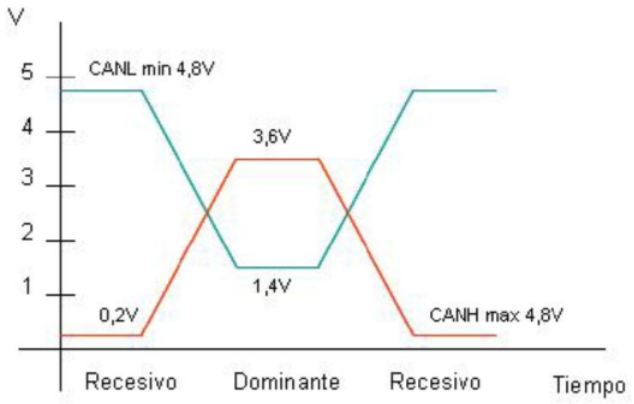


Figura 2: Forma de la trama diferencial del bus CAN para la especificación ISO 11519.[2]

Los nodos conectados a la red CAN de baja velocidad y tolerante a fallos, dispondrán de los siguientes voltajes:

- Dominante: El voltaje diferencial $CAN_H - CAN_L$ es de $2V$ con $CAN_H = 3.5V$ y $CAN_L = 1.5V$.

- Recesivo: El voltaje diferencial $CAN_H - CAN_L$ es de $5V$ con $CAN_H = 0V$ y $CAN_L = 5V$.

A diferencia del estándar de alta velocidad, es necesario que cada uno de los dispositivos de los nodos CAN esté conectados a una resistencia de 120 ohm, para reducir la velocidad de transmisión y sean detectables los fallos en la red.

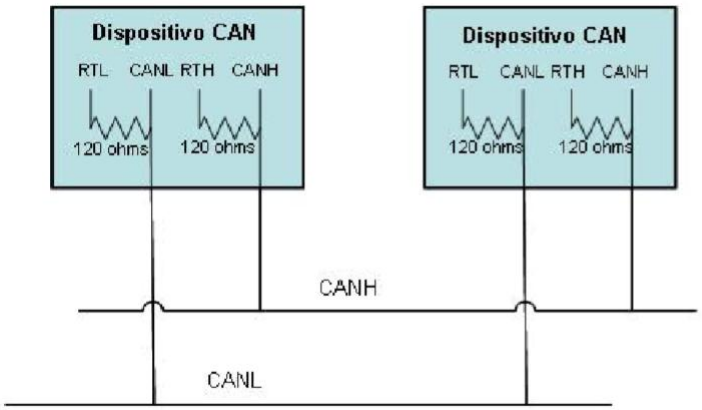


Figura 3: Nodo de bus CAN de baja velocidad. [2]

Estándar 11898:

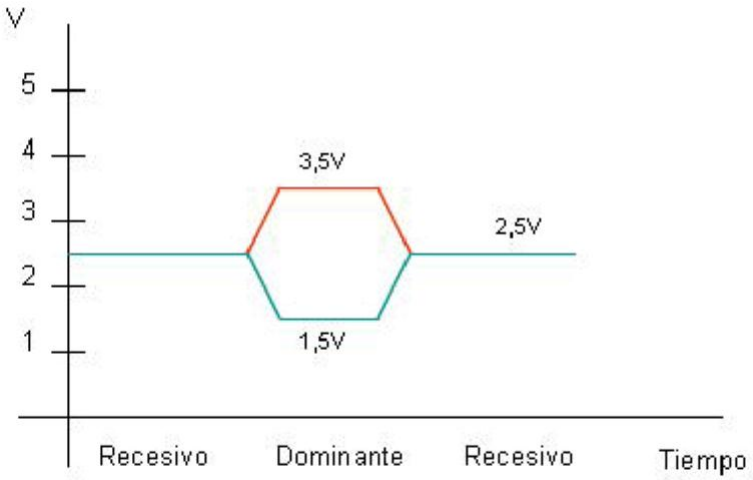


Figura 4: Forma de la trama $CAN_L - CAN_H$ para la especificación ISO 11898.[2]

Los nodos de la red CAN de alta velocidad, podrán recibir las siguientes señales lógicas:

- Dominante: El voltaje diferencial $CAN_H - CAN_L$ es de $2V$ con $CAN_H = 3.5V$ y $CAN_L = 1.5$.
- Recesivo: El voltaje diferencial $CAN_H - CAN_L$ es de $0V$ con $CAN_H = CAN_L = 2.5V$.

Adicionalmente, la red CAN debe estar conectada cada extremo a una resistencia de 120 ohm, en vez de conectarlos a cada transmisor de cada nodo CAN ya que limita la velocidad de transmisión. De esta forma se alcanza a 1Mbps de velocidad de transmisión, no obstante, los errores de transmisión a este nivel deberán ser controlados por otros niveles o capas.

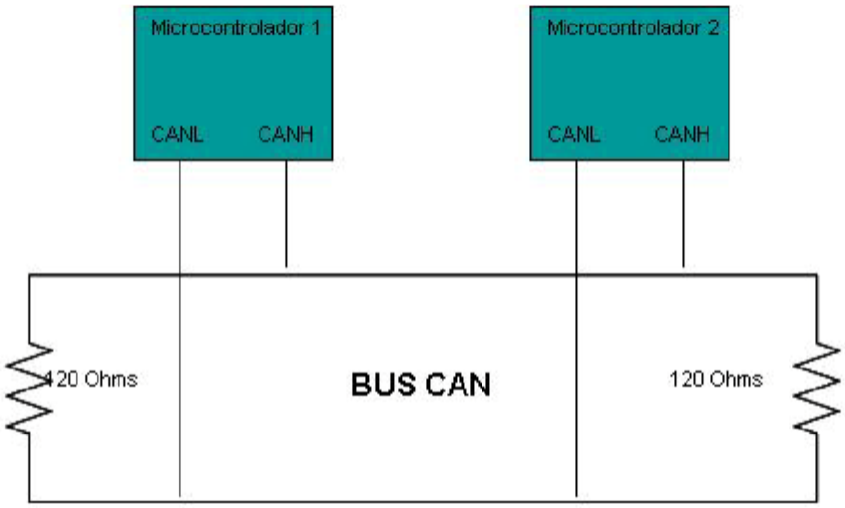


Figura 5: Topología CAN de alta velocidad. [2]

Otro componente crucial de la capa física es la velocidad de transmisión del bus CAN. Como en la mayoría de los sistemas físicos, la longitud del par por el cual transmitimos influye en la velocidad de transmisión, teniendo una relación proporcionalmente inversa entre distancia y velocidad; es decir, un cable más corto tendrá mayor velocidad de transmisión de datos que un cable largo. En la siguiente tabla se muestran los valores típicos de transmisión en un bus can con respecto a la longitud del cable.

Velocidad	Tiempo de Bit	Longitud Máxima
1 Mbps	1 µS	30 m
800 Kbps	1.25 µS	50 m
500 Kbps	2 µS	100 m
250 Kbps	4 µS	250 m
125 Kbps	8 µS	500 m
50 Kbps	20 µS	1000 m
20 Kbps	50 µS	2500m
10 Kbps	100 µS	5000 m

Cuadro 1: Relación Velocidad/Longitud del cable de Bus CAN

La capa de Enlace de datos

La especificación CAN asigna a este nivel la gestión del acceso al medio (MAC, Medium Access Control) y el control lógico (LLC, Logic Link Control). Para ello, define unos mensajes o tramas para la gestión de las comunicaciones y cuyo responsable es la subcapa MAC.

Las tramas definidas por la especificación CAN, son de cuatro tipos y se describen a continuación:

- Trama de datos (Data Frame): Es la trama que generalmente transmite el Bus CAN para él envió de datos.
- Trama remota (Remote Frame): Esta trama es usada por el dispositivo CAN que solicita él envió de datos a los destinatarios.
- Trama de error (Error Frame): Al detectarse un error en el bus CAN, se transmite esta trama desde el dispositivo que lo ha detectado.

- Trama de sobrecarga (Overload Frame): Indica que el nodo que la transmite requerirá de un tiempo antes de poder recibir otra trama de datos o remota.
- Espaciado entre tramas: Entre trama y trama transmitida, se transmite una secuencia predefinida para establecer el final de una trama.
- Bus en reposo: Para que el bus este en escucha de nuevas tramas, se mantiene el bus al nivel recesivo constantemente.

La trama de datos y la remota están construidas por una secuencia de bits de longitud mínima de 44 bits en el formato estándar. En la especificación CAN, establecen grupos para la definición de la cabecera de la trama transmitida, como se muestra en la siguiente tabla:

Cabecera	Longitud	Descripción
Start of Frame (SoF)	1 bit	Indica el comienzo de la trama de datos estándar.
Identifier	11 bits	Representa el identificador del mensaje y la prioridad que tiene.
Remote Transmission Request	1 bit	Esta cabecera indica si es una trama de datos (Valor a 0) o si es una trama remota (valor a 1).
Identifier Extensión bit	1 bit	Indica si la trama es en formato estándar (0) o si es extendida (1).
Reserved bit	1 bit	Establecido a (0), no se usa.
Data Length Code	4 bits	Indica en número de octetos que están disponibles en el campo de datos.
Data Field [3]	0 - 8 bits	Los bytes de datos que incorpora el frame.
CRC	15 bits	Checksum de la trama.
CRC delimiter	1 bit	Delimita el CRC con el resto de datos de la cabecera. Debe tener el valor (1).
ACK slot	1 bit	Indica que al menos un nodo ha recibido correctamente la trama.
ACK delimiter	1 bit	Separa el ACK slot del resto de datos de la trama. Debe tener el valor (1).
End of Frame (EoF)	1 bit	Indica la finalización de la trama.

Cuadro 2: Secuencia de 44 bits de trama CAN. [2]

Gestión de acceso al bus

En el momento de la transmisión, si dos o más nodos CAN coinciden en la transmisión el arbitraje del acceso al bus concederá al nodo con el mensaje más prioritario la escritura sobre él. A continuación se muestra el acceso simultáneo de 3 nodos al bus CAN.

Cada uno de los nodos, transmite bit a bit su mensaje y, para cada bit transmitido, lo compara con el bit recibido. Mientras que el bit transmitido y el bit comparado sean el mismo, el nodo que transmite lo seguirá realizando. No obstante, si el bit difiere, el nodo pierde el acceso al bus, y espera a un espaciado entre tramas para volver a intentar escribir sobre el bus de nuevo.

Se ha de considerar, para que esto suceda, la transmisión del valor “0” (dominante) prevalece antes que el valor “1” (recesivo) y, por ello, si un primer nodo transmite un “1” y otro un “0”, el segundo nodo gana el arbitraje y seguirá transmitiendo, mientras que el primer nodo tendrá que esperar a que termine esta transmisión para volver a realizarla.

[3] En una trama remota el campo DATA FIELD no incluye datos.

De esta forma, el mensaje con un identificador expresado en decimal más bajo, es el de mayor prioridad ya que contiene un mayor número de valores "0" en las posiciones de más peso del mensaje o trama CAN.

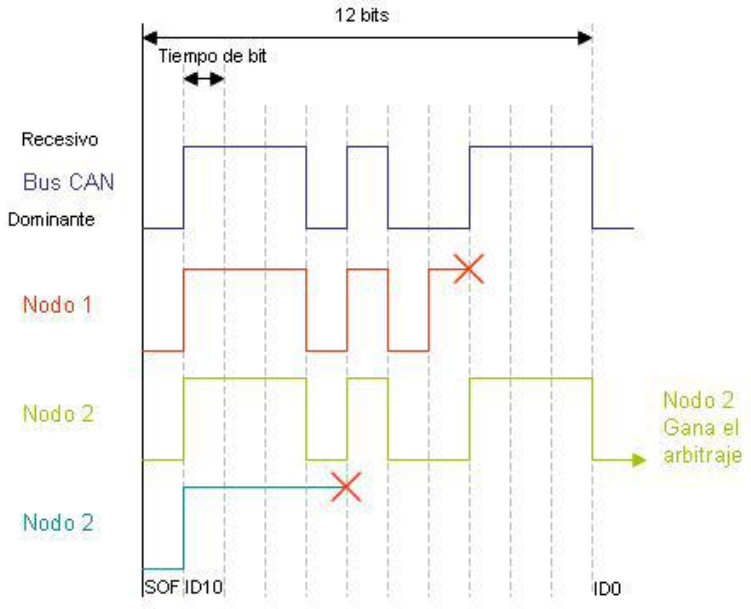


Figura 6: Arbitraje del Bus CAN. [2]

Por otro lado, cabe destacar que el nodo que transmite el mensaje más prioritario, no se le requiere la retransmisión de su mensaje, después de haber terminado el arbitraje. Así pues, continuará con su transmisión del resto de datos que le restaban por transmitir consiguiendo así, un uso eficiente del ancho de banda del bus CAN.

Detección y Gestión de Errores

La capa MAC contribuye con las labores de control y detección de errores sobre el bus CAN. Por un lado, se gestionan los errores a nivel de mensaje gracias a los mecanismos que se han establecido en la especificación y, por otro, se gestionan los errores a nivel de nodo ya que, si un nodo no funciona correctamente o ha provocado un número elevado consecutivo de errores durante la transmisión de un mensaje, éste se auto aísla del resto de nodos para evitar el mal funcionamiento del bus.

Los errores a nivel de mensaje pueden ser los siguientes:

- Error de bit (Bit Error): Se produce este error cuando el nodo detecta que *(Freescale, 2012, p. 601)*durante la transmisión de su mensaje, el bit transmitido no coincide con que se ha recibido.
- Error de relleno (Stuffing Error): Este error se produce cuando se detectan 6 bits consecutivos del mismo valor en las cabeceras de la trama CAN, que deban seguir el "Bit Stuffing".
- Error de CRC: Cuando el Checksum que calcula el receptor, no coincide con el Checksum del mensaje recibido, se produce este error.
- Error de forma: Se produce cuando se ha recibido una trama con una cabecera de tamaño fijo, con otra longitud en bits diferente.
- Error de reconocimiento (ACK Error): Se produce cuando ningún nodo receptor escribe sobre campo "ACK spot".

Un dispositivo puede tener uno de los siguientes tres estados:

- Estado de error activo: El estado de error activo es un estado en el cual la unidad puede participar en comunicaciones sobre el bus de manera normal. Si el dispositivo detecta un error, transmite una bandera de error activo.
- Estado de error pasivo: Aunque un dispositivo en estado de error pasivo puede participar en comunicaciones en el bus, no puede notificar a otras unidades de un error mientras está recibiendo datos de manera que no interrumpa sus comunicaciones. Incluso cuando la unidad en estado de error pasivo ha detectado un error, si los otros dispositivos en el bus no han detectado una unidad en estado de error activo se asume que no ocurrió ningún error en el bus. Cuando el dispositivo en estado de error pasivo ha detectado un error, transmite una bandera de error pasivo; Además un dispositivo en estado de error pasivo no puede iniciar una transmisión inmediatamente después de terminar de enviar un paquete. Un periodo de transmisión de paquete compuesto de 8 bits recesivos es insertado en un espacio entre paquetes antes de que la próxima transmisión pueda iniciar.
- Estado de Bus en reposo: En el estado de bus en reposo el dispositivo no puede comunicarse dentro del bus.

Cada uno de estos estados es administrado mediante el contador de errores de transmisión y el contador de errores de recepción, obteniendo el estado de error relevante mediante una combinación de ambos contadores. La relación entre estados de error y valores de contador se muestra en la siguiente tabla:

ESTADO DE ERROR	CONTADOR DE ERROR DE TRANSMISIÓN Y CONTADOR DE ERROR DE RECEPCIÓN
Error activo	0-127 Y 0-127
Error pasivo	128-255 O 128-255
Bus en reposo	Mínimo 256

Cuadro 3: Estados de error del Bus CAN

La plataforma TRK-MPC5606B

La plataforma TRK-MPC5606B es una tarjeta de desarrollo orientada a la industria automotriz fabricada por Freescale. La tarjeta incluye:

- Microcontrolador MPC5606B en empaquetado 144LQFP.
- Conexión JTAG mediante un circuito OSBDM utilizando el MCU MPC9S08JM.
- Transceiver CAN MCZ3390S5EK.
- Interfaz CAN y LIN.
- Interfaz analógica con potenciómetro.
- LEDs de alta luminosidad.
- Interfaz de comunicación serial SCI.

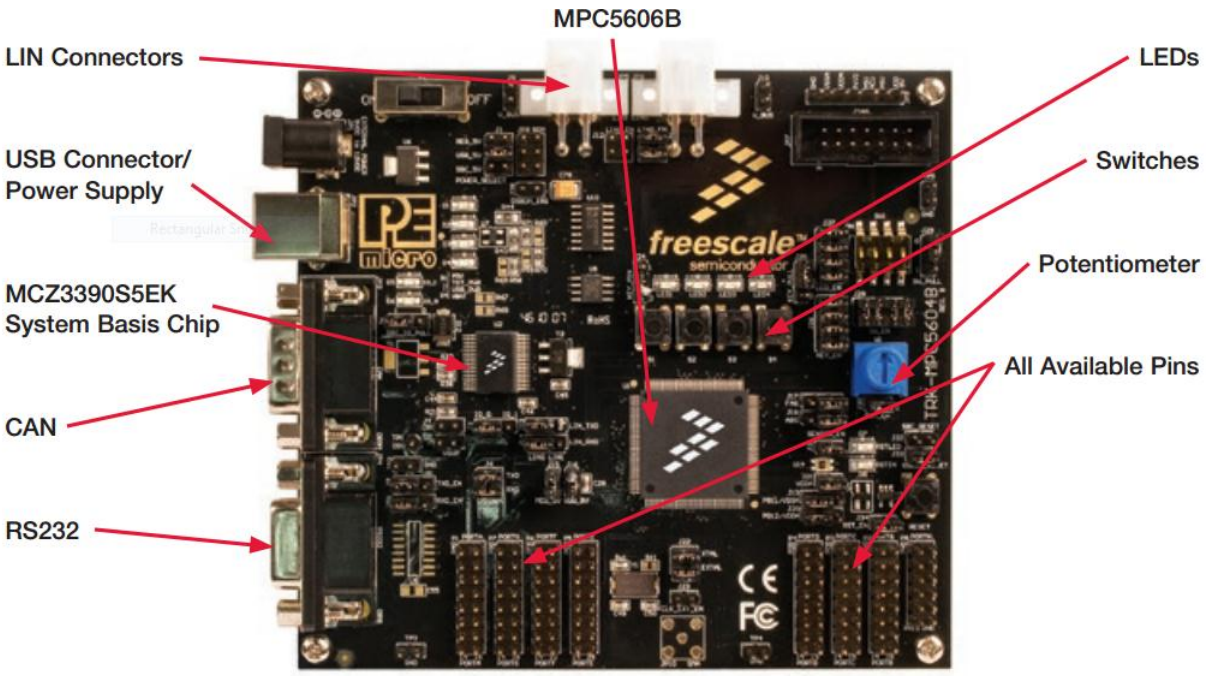


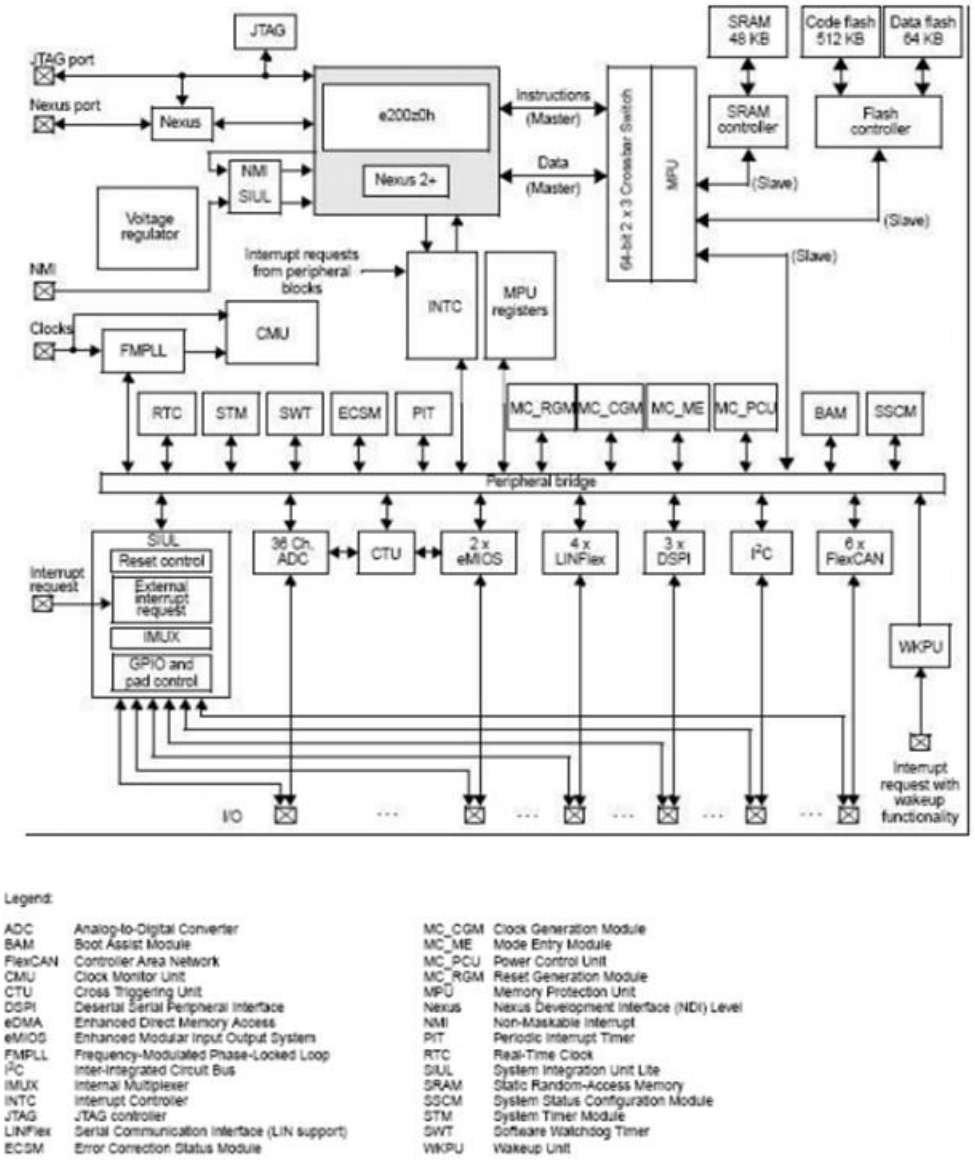
Figura 7: Tarjeta TRK-MPC5606B.[3]

El Microcontrolador MPC5606B

El MPC5606B es un microcontrolador desarrollado por Freescale Semiconductor, perteneciente a la familia MPC560x, también conocida como Qorivva. Es un microcontrolador de 32 bits dedicado para aplicaciones orientadas a la electrónica de carrocería automotriz. Su núcleo está basado en arquitectura PPC. Sus principales características son:

- 1 MB de memoria flash para código, 64 KB de flash para datos y 80 KB de memoria SRAM.
- Frecuencia de operación del núcleo de hasta 64 MHz, basado en un lazo de seguimiento de fase modulado por frecuencia (FM PLL).
- Controlador de Interrupciones (INTC) con 148 vectores de interrupción prioritarios seleccionables, incluyendo 16 interrupciones externas.
- 32 canales para conversores análogo digital de 10 bits (ADC) y 16 canales para para conversores análogo digital de 12 bits.

- 6 módulos de interfaz serial periférica (DSPI), 8 interfaces de comunicación serial (LINFlex), 6 módulos CAN (FlexCAN).
- 123 entradas y salidas de uso general configurables (GPIO).
- 6 timers periódicos con interrupción (PIT) con una resolución de 32 bits.
- Debugger en el sistema por medio de interfaz JTAG (IEEE 1149.1).



Legend:

ADC	Analog-to-Digital Converter	MC_CGM	Clock Generation Module
BAM	Boot Assist Module	MC_ME	Mode Entry Module
FlexCAN	Controller Area Network	MC_PCU	Power Control Unit
CMU	Clock Monitor Unit	MC_RGM	Reset Generation Module
CTU	Cross Triggering Unit	MPU	Memory Protection Unit
DSPI	Deserial Serial Peripheral Interface	Nexus	Nexus Development Interface (NDI) Level
eDMA	Enhanced Direct Memory Access	NMI	Non-Maskable Interrupt
eMIOS	Enhanced Modular Input Output System	PIT	Periodic Interrupt Timer
FMPLL	Frequency-Modulated Phase-Locked Loop	RTC	Real-Time Clock
I²C	Inter-Integrated Circuit Bus	SIUL	System Integration Unit Lite
IMUX	Internal Multiplexer	SRAM	Static Random-Access Memory
INTC	Interrupt Controller	SSCM	System Status Configuration Module
JTAG	JTAG controller	STM	System Timer Module
LINFlex	Serial Communication Interface (LIN support)	SWT	Software Watchdog Timer
ECSM	Error Correction Status Module	WKPU	Wakeup Unit

Figura 8: Diagrama de bloques del MPC56060B. [4]

El módulo FlexCAN

El MPC5606B contiene 6 módulos FlexCAN, llamados CAN_0, CAN_1, CAN_2, CAN_3, CAN_4 y CAN_5. Los módulos FlexCAN implementan el protocolo CAN de acuerdo a la especificación CAN 2.0B. Los paquetes pueden tener un tamaño de hasta 8 bytes y una velocidad de transmisión de hasta 1 Mbit/s.

El diagrama de bloques que se muestra a continuación describe la estructura de un módulo FlexCAN. La memoria SRAM está dedicada al almacenamiento de los buffers de mensaje (MBS, Message Buffer Storage) y a las ID de los registros de almacenamiento de máscaras. Hasta 64 mensajes pueden ser almacenados en cualquier momento dado. Cada uno de estos 64 buffers de mensaje almacena configuración y datos de control, time stamps, IDs de mensaje y los a enviar o transmitir. Los 64 buffers de un módulo FlexCAN son llamados BUF[0] a BUF[63].

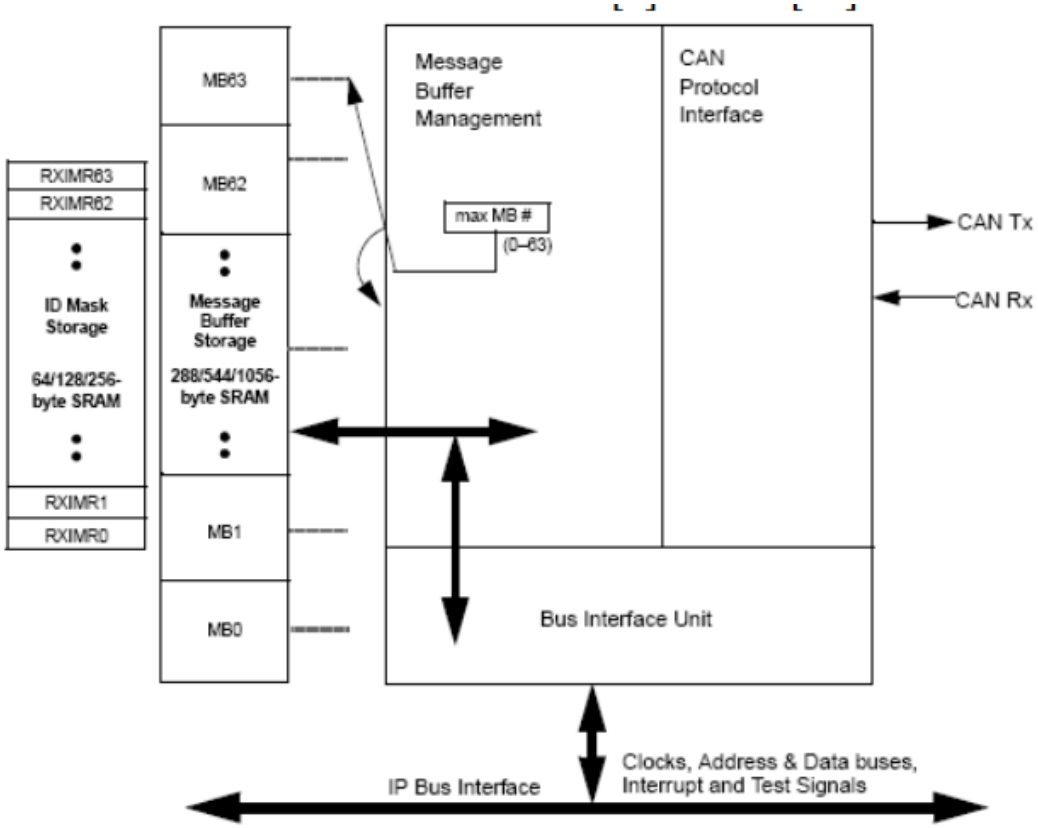


Figura 9: Diagrama de bloques del módulo FlexCAN.[4]

El módulo FlexCAN se compone de 3 sub-módulos:

- La interfaz de protocolo CAN maneja la comunicación serial del bus, la recepción y transmisión de los mensajes, la validación de los mensajes recibidos y el manejo de errores.
- El manejador de Buffers de Mensaje se encarga de seleccionar el buffer para la transmisión y recepción.
- La unidad de interfaz de bus controla el acceso al bus interno de conexión con la CPU. La interfaz recibe datos y un pulso de reloj de la CPU y provee datos recibidos e interrupciones.

El módulo de FlexCAN tiene 5 modos de operación:

- Modo normal (usuario o supervisor): El módulo está activo, envía y recibe mensajes normalmente y todas las funciones del protocolo CAN están activadas.
- Modo Freeze: En este modo, ningún proceso de transmisión o recepción está permitido. El módulo pierde sincronización con el bus CAN. Si se activa el bit FRZ, el módulo entrara en modo Freeze si el bit HALT está activado en el registro MCR o si el módulo está en modo debug.
- Modo de solo recepción: La transmisión se desactiva, el módulo solo recibe mensajes reconocidos por otra estación CAN (Por lo tanto debe de haber al menos otros dos dispositivos en el bus CAN para que este modo funcione).

- Modo de loopback: Se crea un lazo interno entre la salida del transmisor y la entrada del receptor para poder realizar pruebas de funcionamiento del Transceiver. El módulo entra en este modo solo si el bit LPB en el registro MCR se encuentra en 1.
- Modo de bajo consumo de energía: Todos los pulsos de reloj se apagan. El módulo entra en este estado si el bit MDIS en el registro MCR se activa, siempre y cuando todos los procesos de transmisión y recepción en curso hayan terminado.

La tarjeta incluye un conector DB9 (JP3) que enlaza a los pines CANH y CANL del Transceiver SBC MCZ3390S5EK (U2). Los pines TX y RX del transceiver están conectados a los pines PC11 y PC10 del microcontrolador; El pin PC10 está asociado a CAN1TX (Función Alternativa AF1) y PC11 puede ser asociado a CAN1RX o CAN4RX (activados mediante el registro PSMI en el SIUL). Para que el transceiver inicie operación se deberá forzar su entrada en modo debug. Para poder activar el modo debug se deberá de cumplir las siguientes condiciones:

- El SBC deberá ser alimentado con un voltaje máximo de 12V. En la tarjeta se deberá de conectar una fuente de poder externa, así como poner en corto los pines 5 y 6 del conector J1. Para verificar el funcionamiento adecuado el pin 16 del SBC (DBG) debe tener un voltaje entre 8 y 10 V, el pin 22 (RST) este a 5 V y el pin 6 (5VCAN) este a 5V (Prueba que el regulador de voltaje CAN está funcionando de manera apropiada).
- En modo debug el transceiver CAN siempre está activo, para forzar el dispositivo a entrar a este modo el microcontrolador deberá enviar los comandos SPI adecuados al transceiver. En la tarjeta, los pines del módulo DSPI1 del microcontrolador (Puerto H) están conectados al SBC.[9]

La tarjeta Arduino Mega 2560

El Arduino Mega 2560 es un microcontrolador basado en el ATmega2560. Cuenta con 54 GPIOs (de los cuales 15 se pueden usar como salidas PWM), 16 entradas analógicas, 4 UARTs, un oscilador de cristal de 16 MHz, un conector USB, una entrada de alimentación, un conector ICSP y un botón de Reset. Contiene todo lo necesario para simplemente conectar la tarjeta a una computadora mediante USB y cargar el programa para que el microcontrolador pueda funcionar. El Arduino podrá ser programado en ASM compatible con AVR, C, o el lenguaje propietario de Arduino, un conjunto de librerías escritas en C y ASM, el cual se asemeja a C++. El ATmega2560 cuenta con 256 KB de memoria flash para almacenar código (de los cuales 8KB se utilizan para almacenar el bootloader de Arduino), 8 KB de SRAM y 4 KB de EEPROM.

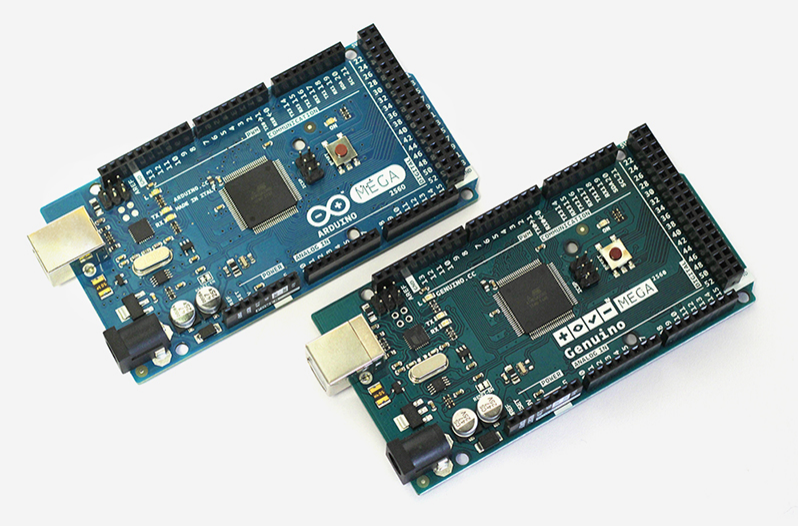


Figura 10: Arduino Mega 2560 [5]

El CAN-BUS Shield

Los Shields son circuitos que siguen la metodología modular del Arduino. Estos circuitos se conectan en una topología estilo torre, la cual permite conectar múltiples shields a un solo microcontrolador. La restricción de esta topología esta en los pines de chip select para el bus SPI de cada shield, ya que el bus SPI exige un pin CS por cada dispositivo conectado al bus.

El CAN-BUS Shield incluye un controlador para CAN Bus MCP2515 de Microchip con una interfaz SPI y un transceiver CAN MCP2551 de Microchip; Este shield le da a la tarjeta Arduino la capacidad de comunicarse con otros dispositivos a través de una red CAN 2.0.

Figura 11: CAN-BUS Shield de Seeed Studio[6]

Objetivos

Objetivo General

Generar una aplicación para micro controlador Freescale, que, mediante el protocolo de comunicación CAN bus 2.0B, envié información acerca de un proceso, interactuando con su instrumentación y actuadores.

Objetivos Específicos

- Generar una plataforma de instrumentación y actuadores para el proceso.
- Generar un servidor para el monitoreo e interacción del usuario con el proceso.
- Establecer una red CAN bus 2.0B que comunique la tarjeta de control (Freescale TRK-MPC5606B) con la tarjeta de monitoreo (Arduino Mega 2560).
- Fortalecer el desarrollo académico de materias de la carrera de Ingeniero En Electrónica y Comunicaciones como Comunicaciones, Instrumentación y Microcontroladores, generando metodologías y conocimientos que sienten un precedente para el nivel de las practicas realizables en estas materias.

Metodología

Infraestructura

Para la implementación del proyecto se utilizara la tarjeta TRK-MPC5606B de Freescale, la tarjeta Arduino Mega 2560 con el CAN-BUS Shield de Seeed Studio, el analizador de protocolo CAN BUS Analyzer Tool AGPDT002 de Microchip, el software Freescale Codewarrior para la programación del microcontrolador y el software Microchip CAN BUS Analyzer para observar los paquetes enviados en el bus CAN. Se desarrollara una aplicación para el control de un VFD (Variable Frequency Drive) Parker AC690+, el cual controla un motor trifasico de corriente alterna.

Implementación

Software RAppID

Primero se procederá a la elaboración del código. Un microcontrolador como lo es el MPC5606B contiene una gran cantidad de registros y periféricos, por lo que Freescale ha desarrollado una herramienta que nos permite generar el código driver necesario para la operación de cada módulo, llamada RAppID. A continuación se detalla el proceso para inicializar un programa de uso de protocolo CAN para la tarjeta TRK-MPC5606B: Comenzamos por abrir el programa RAppID Pin Wizard, donde seleccionamos la opción MPC5606B y damos clic en "Start Wizard". Después nos pedirá seleccionar el empaquetado del circuito integrado. Seleccionamos la opción "144 QFP" y damos clic en "Next". Después configuraremos los pines del microcontrolador como se muestra en la tabla:

Function	Input	Output	User Assigned Signal Name
ADC_0_ADC_1 ANP0	PB4		ANP0_Potentiometer_Input

Cuadro 4: Configuración de pines ADC en el software RAppID

Function	Input	Output	User Assigned Signal Name
DSPI_1 Chip Select 0		PH3	PH3_DSPI1_CS0_Output
DSPI_1 Clock		PH2	PH2_DSPI1_CLK_Output
DSPI_1 Data Out		PH1	PH1_DSPI1_Data_Output
DSPI_1 Data In	PH0		PH0_DSPI1_Data_Input

Cuadro 5: Configuración de pines DSPI1 en el software RAppID

Function	Input	Output	User Assigned Signal Name
CAN_1 Tx		PC10	PC10_CAN1_Tx
CAN_1 Rx	PC11		PC11_CAN1_Rx

Cuadro 6: Configuración de pines CAN1 en el software RAppID

Function	Input	Output	User Assigned Signal Name
PE4		PE4	PE4_LED1_Output
PE5		PE5	PE5_LED2_Output
PE6		PE6	PE6_LED3_Output
PE7		PE7	PE7_LED4_Output

Cuadro 7: Configuración de los LEDs en el software RAppID

Hemos terminado con la ubicación de pines. Ahora presionamos "Next" tres veces para llegar a la ventana de "System Clock/Timers". Aquí modificaremos las siguientes opciones:

- En la pestaña Mode Entry
    - Subpestaña General Configuration:
        - En el apartado Mode Configuration buscar el modo DRUN y seleccionar en la lista de la última columna la opción System PLL
    - Subpestaña Peripheral Configuration
        - Seleccionar el botón marcado como "Normal"
- En la pestaña SWT
    - Desactivar la opción "Enable Watchdog Timer"

A continuación damos clic en "Next" para terminar la configuración de Timers. Se nos presenta ahora la ventana de configuración de Periféricos, la cual se basa en la selección de pines que hicimos en el procedimiento anterior. Para comenzar seleccionamos la opción DSPI.
En la siguiente ventana seleccionaremos las siguientes opciones:

- Master/Slave Mode: Master
- Peripheral Chip Select Line 0 inactive state: High
- Halt Mode: Disable
- Presionar "Next"

Después, configuraremos el módulo ADC, seleccionando la opción ADC. En la ventana, dentro de la pestaña superior «ADC_0» seleccionaremos las siguientes opciones

- En la pestaña «Device Setup», desactivamos la opcion «Power Down Enable».
- En la pestaña «Channel Setup», Activamos el CH0 en modo «Normal».
- Presionamos «OK».

Ahora configuraremos el módulo FlexCAN, seleccionando la opción FlexCAN. En la ventana de FlexCAN seleccionaremos las siguientes opciones:

- Module: Enable
- Freeze Enable: Disable
- Halt FlexCAN: Disable
- Clock Source: System
- CAN Speed (Kbits/s): 500
- Presionamos "OK"

Por ultimo realizaremos la configuración de EMIOS. En esta ventana seleccionaremos las siguientes opciones (Ilustración 21):

- Pestaña "Module Configuration"
    - Seleccionar Global Time base Enable
    - Seleccionar Enable Global Prescaler
    - Seleccionar "OK"

Con esto concluimos la configuración de periféricos y podemos pasar a generar el código. Presionamos "Exit Wizard". En la siguiente ventana seleccionamos el menú "Configuration" y de ahí la opción "Code Generation". Nos aseguramos que las opciones seleccionadas sean las mismas que se ven en la ilustración, y en el apartado de "Source Path" seleccionamos donde se generara el código.

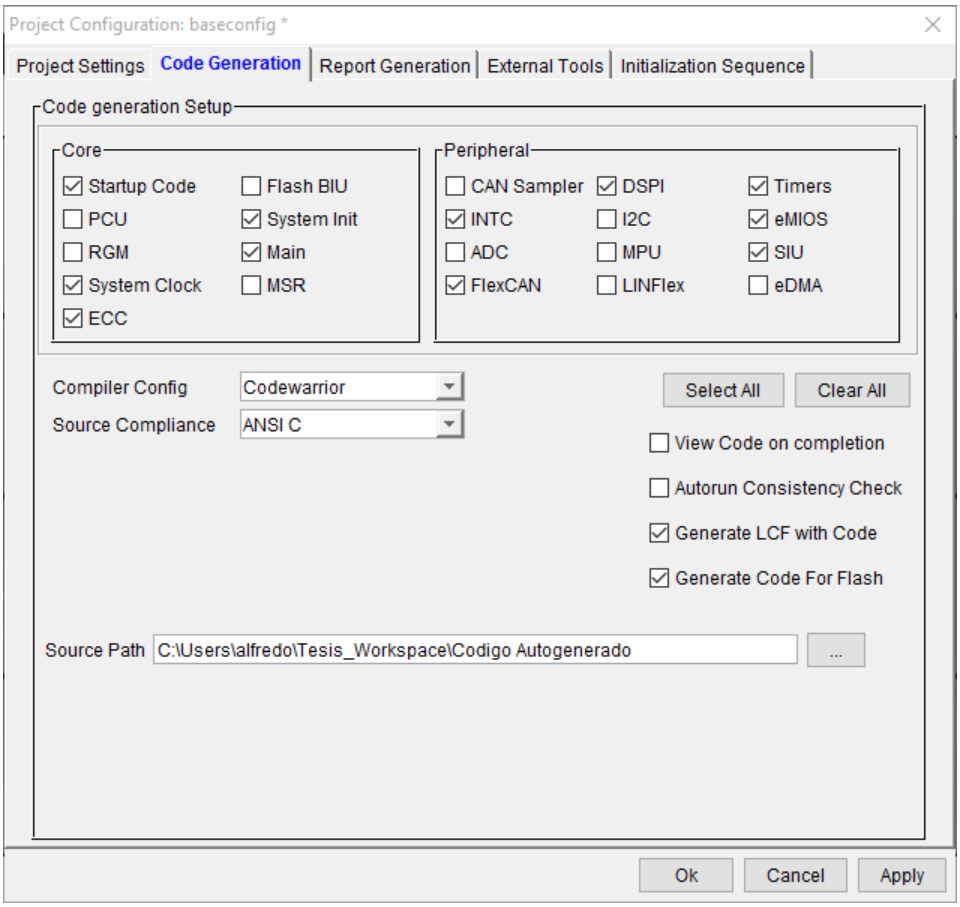


Figura 12: Opciones de generación del código

Ahora en el menú "View" seleccionamos "View Section Map". Ahí cambiaremos la opción "Select Target" de "RAM" a "FLASH" y seleccionamos "OK".

Ahora estamos listos para generar el código. En el menú "Code/Report Generation" seleccionamos la opción "Generate Source Code". Se nos pedirá guardar el proyecto sobre el que hemos estado trabajando; Se recomienda guardarlo en un lugar fácil de recordar ya que el archivo se volverá a utilizar en los siguientes apartados del documento. Después de guardar el archivo el código se habrá generado y habremos terminado de utilizar RAppID por el momento.

Software Freescale CodeWarrior

Al iniciar el software CodeWarrior seleccionamos el menú "File\New\Bareboard Project". A continuación, se creara una ventana nueva, donde en primera instancia seleccionaremos el nombre del proyecto.Al presionar "Next", en la siguiente ventana escribiremos en el cuadro de texto que pone "type filter text" el código "MPC5606B" para seleccionar el microcontrolador a usar en el proyecto, después seleccionamos "Next" y en la siguiente ventana "Finish". Una vez creado el proyecto procedemos a cambiar la compilación del proyecto de RAM a Flash. Ahora cerramos el proyecto en CodeWarrior pero sin cerrar el programa, y regresamos a RAppID.
En RAppID, en el menú "External Tools" Seleccionamos la opción "CWPjmaker", donde se abrirá la siguiente ventana:

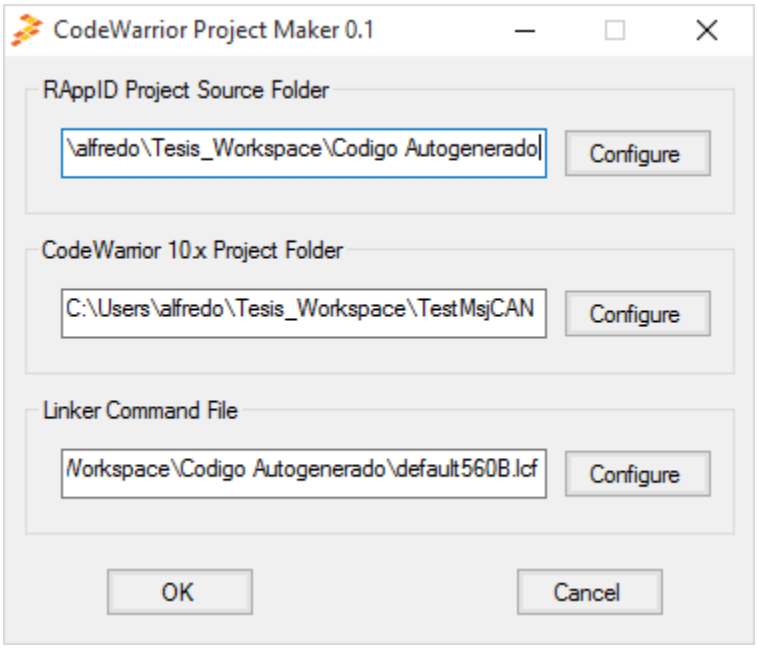


Figura 13: CodeWarrior Project Maker

Seleccionaremos entonces la carpeta del código que generamos en la sección anterior con RAppID, en el siguiente cuadro, la carpeta del proyecto de CodeWarrior, y para el tercer cuadro de texto, seleccionaremos el archivo "default560B.lcf" que se encuentra en la carpeta del código auto generado. A demás de esto copiaremos en la carpeta del proyecto de CodeWarrior, en la subcarpeta "Sources" todo el contenido de la carpeta "C:\Freescale\FastStartKit\DriverCode\TRK5606B", quitando también desde el explorador de Windows la propiedad de Solo Lectura de todos los archivos incluidos en la carpeta. Al reabrir el proyecto en CodeWarrior podemos ver que los archivos se han incluido en el proyecto.

Ahora desde CodeWarrior seleccionaremos la opción "Build Settings", que abrirá la ventana que se muestra a continuación. Ubicaremos dentro del apartado "C/C++ General" el sub-apartado "Paths and Symbols" y agregaremos la entrada "${ProjDirPath}/Sources/" (Sin comillas) en la opción "Assembly Source File" . Al terminar este paso estaremos listos para pasar al código del programa.

Codigo del programa

Se ha de desarrollar una aplicación que corresponda al siguiente diagrama de flujo:

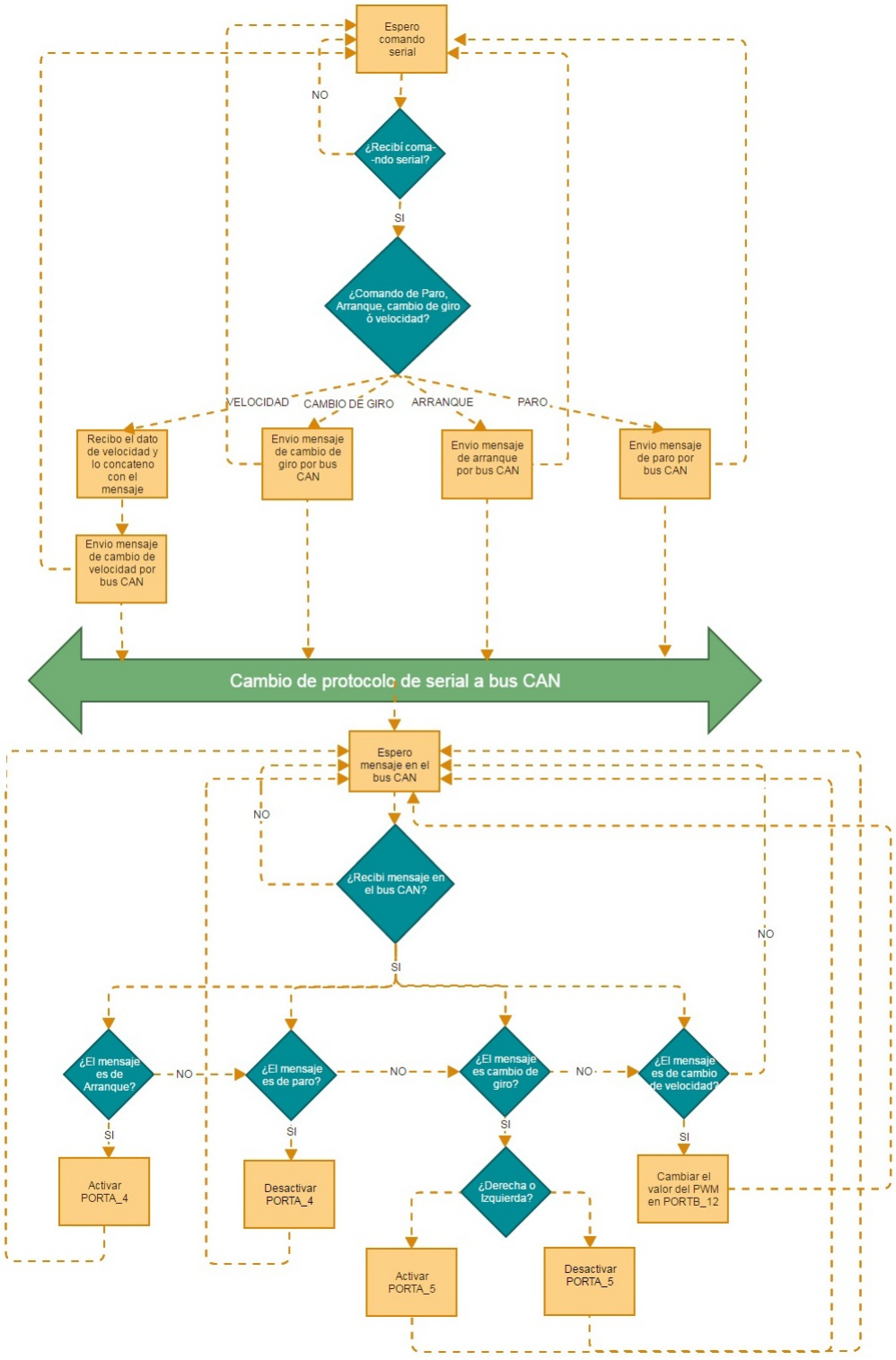


Figura 14: Diagrama de flujo

El código para el archivo “main.c” será el siguiente:

```
/* ------------------------------------------------------------------ */
/*      Control de VFD Parker AC690+ mediante paquetes CAN            */
/*                  Autor: Alfredo Valdés Cárdenas                    */
/* ------------------------------------------------------------------ */

/********************  Archivos de Dependencias **********************/

#include "rappid_ref.h"
#include "rappid_utils.h"
#include "sys_init.h"
#include "CANapi.h"
#include "sbc_hld.h"
#include "gpio_drv.h"
#include "pot_hld.h" purple
#include "Puertos.h"


/*********************  Prototipos de Funciones **********************/

void main(void);
void ProcessCANRX(void);
void Delay(int);
void OutPWM(unsigned long, int);
void initEMIOS_0(void);
void initEMIOS_0ch4(void);
void initGPIO(int,char);
void IOWrite(int,int);
int IORead(int);

/***********************  Variables Globales ************************/

int velocidad = 0;
char direccion = 'd';
int i=0;
uint32_t valor =0;

/* Mensajes CAN que se transmitiran*/

uint8_t msjACK[8] = {1,0,1,0,1,0,1,0};
uint8_t msjError[8] = {0,0xFF,0,0xFF,0,0xFF,0,0xFF};
uint8_t msjVel[8] = {5,6,1,0,1,0,1,0};

/********************  Metodo Principal  ************************/
void main(void) {
/*-----------------------------------------------------------------*/
/*          Funciones de Incializacion del Sistema                 */
/*-----------------------------------------------------------------*/

    sys_init_fnc();
    SBC_Init_DBG();
    initEMIOS_0();
    initEMIOS_0ch4();
    initGPIO(PORTA4,'o');
    initGPIO(PORTA5,'o');
```

```
    /********* Activar Interrupciones Externas *********/

    EnableExternalInterrupts();

    /* Inicializar filtro CAN */
    SetCanRxFilter(0xA, 0, 0);
    CanTxMsg (0x0, 1, 8, (uint8_t *)msjACK, 0);
    while(1){
        ProcessCANRX();
    }
}

void ProcessCANRX(void) {
    can_msg_struct msgCanRX;
    if (CanRxMbFull(0) == 1)/* revisar si se recibieron mensajes CAN*/
    {
        msgCanRX = CanRxMsg(0);//Copiar mensaje recibido del buffer a una
            variable
        switch(msgCanRX.data[0]){//Comparar el primer byte. En base al
            codigo ASCII
        //A es arranque, C es cambio de giro, P es paro y V es velocidad
        case 0x41: //0x41 es A en ASCII, Rutina de comando de Arranque
            IOWrite(PORTA4,1); //Encender PA0, activa el relevador de
                arranque
            CanTxMsg (0x54,1,8,(uint8_t *)msjACK,0); //envia ACK desde 0
                x54, ASCII para T de transmisor
            Delay(1000); //Una pausa de 1 segundo
            break;
        case 0x43:
            if(direccion=='d'){
                IOWrite(PORTA5,1); //Encender PA1, activa el relevador de
                    cambio de giro
                CanTxMsg (0x54,1,8,(uint8_t *)msjACK,0); //envia ACK
                    desde 0x54, ASCII para T de transmisor
                Delay(1000); //Una pausa de 1 segundo
                direccion='i';
            }
            else if(direccion=='i'){
                IOWrite(PORTA5,0); //Encender PA1, activa el relevador de
                    cambio de giro
                CanTxMsg (0x54,1,8,(uint8_t *)msjACK,0); //envia ACK
                    desde 0x54, ASCII para T de transmisor
                Delay(1000); //Una pausa de 1 segundo
                direccion='d';
                }
            break;
        case 0x50: //0x50 es P en ASCII, Rutina de comando de Paro
            IOWrite(PORTA4,0); //Encender PA2, activa el relevador de
                paro
            CanTxMsg (0x54,1,8,(uint8_t *)msjACK,0); //envia ACK desde 0
                x54, ASCII para T de transmisor
            Delay(1000); //Una pausa de 1 segundo
            break;
        case 0x56: //0x56 es V en ASCII, Rutina de comando de Velocidad
            valor=(msgCanRX.data[1]*19999)/100;
            OutPWM(valor,4); //Asignamos velocidad a PE7
```

```
            CanTxMsg (0x54,1,8,(uint8_t *)msjVel,0);//envia ACK desde 0
                x54, ASCII para T de transmisor
            Delay(1000);
            break;
        default :
            CanTxMsg (0x54,1,8,(uint8_t *)msjError,0);//envia Error desde
                0x54, ASCII para T de transmisor
            break;
        }
    }
}
void Delay(int T){
    int i=0;
    int tiempo = T*10000;
    for(i=0;i<tiempo;i++){  }
}
void initEMIOS_0(void) {
    EMIOS_0.MCR.B.GPRE= 1;/* Divide el reloj de 64MHz a 1 MHz*/
    EMIOS_0.MCR.B.GPREN = 1;/* Activar reloj de EMIOS */
    EMIOS_0.MCR.B.GTBE = 1;/* Activar timer global*/
    EMIOS_0.MCR.B.FRZ =1;/*Desactiva los canales de EMIOS en modo debug*/
}
void initEMIOS_0ch4(void) {/*EMIOS0 CH4: Output Pulse Width Modulation*/
    EMIOS_0.CH[4].CADR.R = 0;
    EMIOS_0.CH[4].CBDR.R = 1500;
    EMIOS_0.CH[4].CCR.B.BSL = 0x01;/*Usa el contador "B" del canal EMIOS
        */
    EMIOS_0.CH[4].CCR.B.EDPOL = 1;/* Modo de flanco positivo*/
    EMIOS_0.CH[4].CCR.B.MODE = 0x60;/* Modo OPWM*/
    SIU.PCR[28].R = 0x0600;/* Activa el pin 28 en modo EMIOS*/
}
void OutPWM(unsigned long DC, int ch){
    EMIOS_0.CH[ch].CADR.R = 19999-DC;/*Estable el inicio del pulso PWM*/
    EMIOS_0.CH[ch].CBDR.R = 19999;/*Fin del pulso a 20 ms*/
}
void initGPIO(int ch, char mode){
    if (mode == 'i'){
        SIU.PCR[ch].R = 0x0100;/*Activa el pin "ch" como entrada*/
    }
    if(mode == 'o'){
        SIU.PCR[ch].R = 0x0200;/*Activa el pin "ch" como salida*/
        IOWrite(ch,0);/*Apaga el pin*/
    }
}
void IOWrite(int ch, int state){
    SIU.GPDO[ch].R = !(state); /*Escribe el estado en el pin "ch"*/
}
int IORead(int ch){
    return SIU.GPDI[ch].B.PDI; /*Leer el pin, pasa por el dato "ch" */
}
```

Crearemos también un archivo con las definiciones de entradas y salidas del microcontrolador llamado «Puertos.h»

```
/* ------------------------------------------------------------ */
/*                  Definicion de pines para MPC5606B           */
/*                   Autor: Alfredo Valdés Cárdenas             */
/* ------------------------------------------------------------ */

#define PORTA0 0
#define PORTA1 1
#define PORTA2 2
#define PORTA3 3
#define PORTA4 4
#define PORTA5 5
#define PORTA6 6
#define PORTA7 7
#define PORTA8 8
#define PORTA9 9
#define PORTA10 10
#define PORTA11 11
#define PORTA12 12
#define PORTA13 13
#define PORTA14 14
#define PORTA15 15
#define PORTB0 16
#define PORTB1 17
#define PORTB2 18
#define PORTB3 19
#define PORTB4 20
#define PORTB5 21
#define PORTB6 22
#define PORTB7 23
#define PORTB8 24
#define PORTB9 25
#define PORTB10 26
#define PORTB11 27
#define PORTB12 28
#define PORTB13 29
#define PORTB14 30
#define PORTB15 31
#define PORTC0 32
#define PORTC1 33
#define PORTC2 34
#define PORTC3 35
#define PORTC4 36
#define PORTC5 37
#define PORTC6 38
#define PORTC7 39
#define PORTC8 40
#define PORTC9 41
#define PORTC10 42
#define PORTC11 43
#define PORTC12 44
#define PORTC13 45
#define PORTC14 46
#define PORTC15 47
#define PORTD0 48
```

```
#define PORTD1 49
#define PORTD2 50
#define PORTD3 51
#define PORTD4 52
#define PORTD5 53
#define PORTD6 54
#define PORTD7 55
#define PORTD8 56
#define PORTD9 57
#define PORTD10 58
#define PORTD11 59
#define PORTD12 60
#define PORTD13 61
#define PORTD14 62
#define PORTD15 63
#define PORTE0 64
#define PORTE1 65
#define PORTE2 66
#define PORTE3 67
#define PORTE4 68
#define PORTE5 69
#define PORTE6 70
#define PORTE7 71
#define PORTE8 72
#define PORTE9 73
#define PORTE10 74
#define PORTE11 75
#define PORTE12 76
#define PORTE13 77
#define PORTE14 78
#define PORTE15 79
#define PORTF0 80
#define PORTF1 81
#define PORTF2 82
#define PORTF3 83
#define PORTF4 84
#define PORTF5 85
#define PORTF6 86
#define PORTF7 87
#define PORTF8 88
#define PORTF9 89
#define PORTF10 90
#define PORTF11 91
#define PORTF12 92
#define PORTF13 93
#define PORTF14 94
#define PORTF15 95
#define PORTG0 96
#define PORTG1 97
#define PORTG2 98
#define PORTG3 99
#define PORTG4 100
#define PORTG5 101
#define PORTG6 102
#define PORTG7 103
#define PORTG8 104
```

```
#define PORTG9 105
#define PORTG10 106
#define PORTG11 107
#define PORTG12 108
#define PORTG13 109
#define PORTG14 110
#define PORTG15 111
#define PORTH0 112
#define PORTH1 113
#define PORTH2 114
#define PORTH3 115
#define PORTH4 116
#define PORTH5 117
#define PORTH6 118
#define PORTH7 119
#define PORTH8 120
#define PORTH9 121
#define PORTH10 122
#define PORTH11 123
```

A su vez, el código que se cargara en la tarjeta Arduino es el siguiente:

```
/* ------------------------------------------------------------------ */
/*          Traductor de comandos seriales a paquetes CAN             */
/*                Autor: Alfredo Valdés Cárdenas                      */
/* ------------------------------------------------------------------ */

/*********************  Archivos de Dependencias **********************/


#include <mcp_can.h>
#include <SPI.h>
#define INT8U unsigned char

/***********************  Variables del programa ************************/

const int SPI_CS_PIN = 9; //Chip Select del SPI en pin 9
MCP_CAN CAN(SPI_CS_PIN);  // Indicamos al transceiver cual
                          // es el pin del chip select de SPI
INT8U Flag_Recv = 0;
INT8U len = 0;
INT8U buf[8];
INT32U canId = 0x000;
char str[20];
String inputString = ""; // String que almacena los datos seriales
boolean stringComplete = false; // comprueba que el string este completo
boolean comandoArranque = false; // comprueba si el comando es de arranque
boolean comandoParo = false; // comprueba si el comando es de paro
boolean comandoGiro = false; // comprueba si el comando es de cambio de giro
boolean comandoVelocidad = false; // comprueba si el comando es de cambio de velocidad
boolean direccionGiro = false; // giro a la derecha = true, a la izquierda = false
int porcentaje = 0; // porcentaje de velocidad
unsigned char msjArranque[8] = {0x50, 0, 0, 0, 0, 0, 0, 0};
unsigned char msjCDG[8] = {0x43, 0, 0, 0, 0, 0, 0, 0};
unsigned char msjParo[8] = {0x41, 0, 0, 0, 0, 0, 0, 0};

/*-------------------------------------------------------------------*/
```

```
/*          Funciones de Incializacion del Sistema          */
/*----------------------------------------------------------*/

void parserSerial(void);
void setup(void);
void loop(void);

void setup()
{
  attachInterrupt(0, MCP2515_ISR, FALLING); // Interrupcion CAN,
                                            // avisa cuando llega un Mensaje
  Serial.begin(1200);
  Serial.setTimeout(10000);
  inputString.reserve(200);
  delay(2000);
  CAN.begin(CAN_500KBPS);
}

void MCP2515_ISR()
{
  Flag_Recv = 1;
}

void loop()
{ Serial.println("Esperando comando...");
  parserSerial();
  if (stringComplete) {
    if (comandoArranque == true) {
      CAN.sendMsgBuf(0xA, 0, 8, msjArranque);
      Serial.print("Recibi comando de arranque");
      comandoArranque = false;
      delay(100);
    }
    if (comandoParo == true) {
      CAN.sendMsgBuf(0xA, 0, 8, msjParo);
      Serial.print("Recibi comando de paro");
      comandoParo = false;
      delay(100);
    }
    if (comandoGiro == true) {
      CAN.sendMsgBuf(0xA, 0, 8, msjCDG);
      Serial.print("Recibi comando de cambio de giro");
      comandoGiro = false;
      delay(100);
      }
      if (comandoVelocidad == true)
    {
      porcentaje = inputString.toInt();
        unsigned char msjVelocidad[8] = {0x56, porcentaje, 0, 0, 0, 0, 0, 0};
        CAN.sendMsgBuf(0xA, 0, 8, msjVelocidad);
        Serial.print("Recibi cambio de velocidad a "+porcentaje+"%");
        delay(100);
        comandoVelocidad = false;
      }
      inputString = "";
      stringComplete = false;
```

```
    }
  }

void parserSerial() {
  while (Serial.available()) {
    char inChar = (char)Serial.read(); //Recibe el nuevo byte
    inputString += inChar; // agregalo al inputString
    // dependiendo del caracter recibido, activa una bandera que indique si es
    // un comando de velocidad, cambio de giro, arranque, paro, o fin de linea
    if (inChar == '\n') {
      stringComplete = true;
    }
    else if (inChar == 'A') {
      comandoArranque = true;
      inputString = "";
    }
    else if (inChar == 'P') {
      comandoParo = true;
      inputString ="";
    }
    else if (inChar == 'C') {
      comandoGiro = true;
      inputString = "";
    }
    else if (inChar == 'V') {
      comandoVelocidad = true;
      inputString ="";
    }
  }
}
```

Placa de control

Parker AC 690+

El Parker AC 690+ es un VFD (Variable Frequency Drive), un dispositivo que nos permite controlar la velocidad de un motor trifasico de corriente alterna. El VFD cuenta con un panel de IO que nos permite controlar el motor de manera remota, sus conexiones se ilustran a continuación:

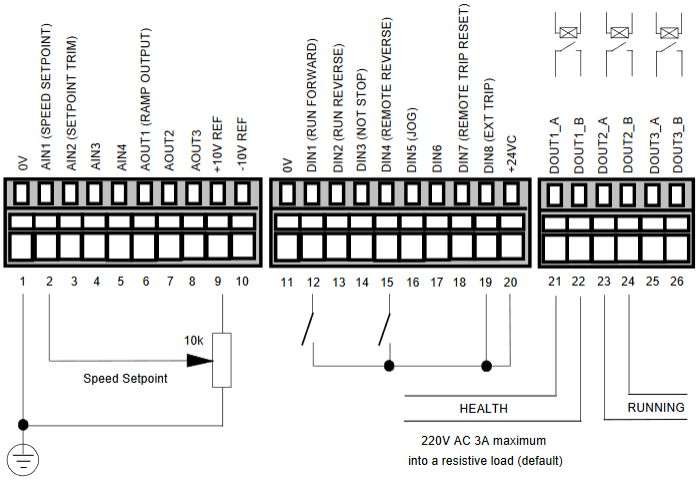


Figura 15: Panel de control del AC 690+[7]

Diseño de la placa de control

Habremos de diseñar una placa que tome las señales del microcontrolador y las convierta a otras que puedan ser interpretadas por el drive. Sera necesario convertir la señal PWM de PORTB_12 a una rampa de voltaje, para ello se utilizara un filtro RC. El calculo apropiado para la resistencia del filtro con frecuencia de corte de 500 hertz, habiendo escogido un capacitor de 1 microfaradio, se lleva acabo con la siguiente ecuación:

$$R_f = \frac{1}{f_c * C_f * 2 * \pi}$$

Donde:

- R_f es la resistencia del filtro.
- $f_c$es la frecuencia de corte, 500 Hz.
- C_f es el capacitor del filtro, de $1\mu F$.

Sustituyendo en la ecuación obtenemos que el valor de la resistencia es:

$$R_f = \frac{1}{500 * .000001 * 2 * \pi} = 318.3099$$

El cual aproximaremos al valor comercial de 330Ω.

También convertiremos las dos señales digitales(PORTA_5 para arranque y paro y PORTA_4 para el cambio de giro) de 5 volts a 24 volts utilizando relevadores. El diagrama esquemático del circuito es el siguiente:

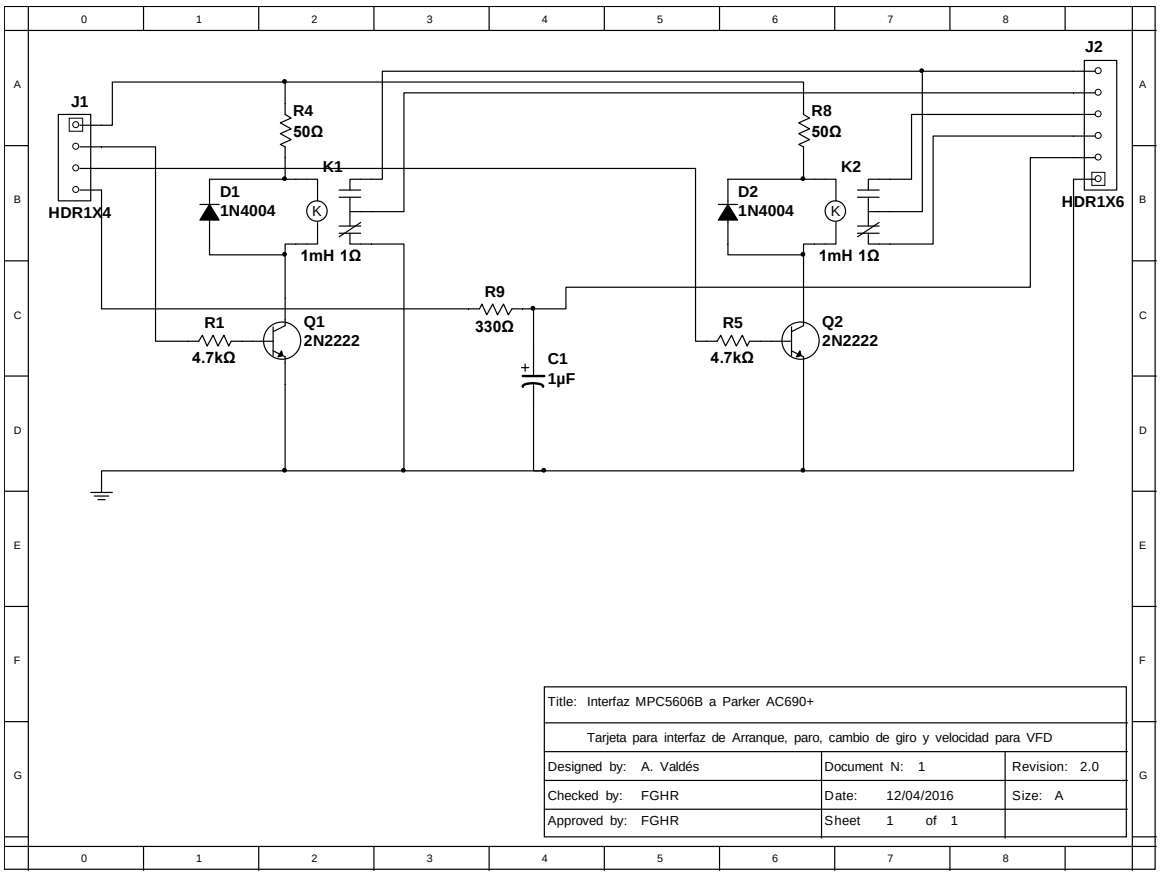


Figura 16: Diagrama esquemático de la tarjeta de control

Se produjo un diseño de PCB para el diagrama utilizando el software Ultiboard de National Instruments.

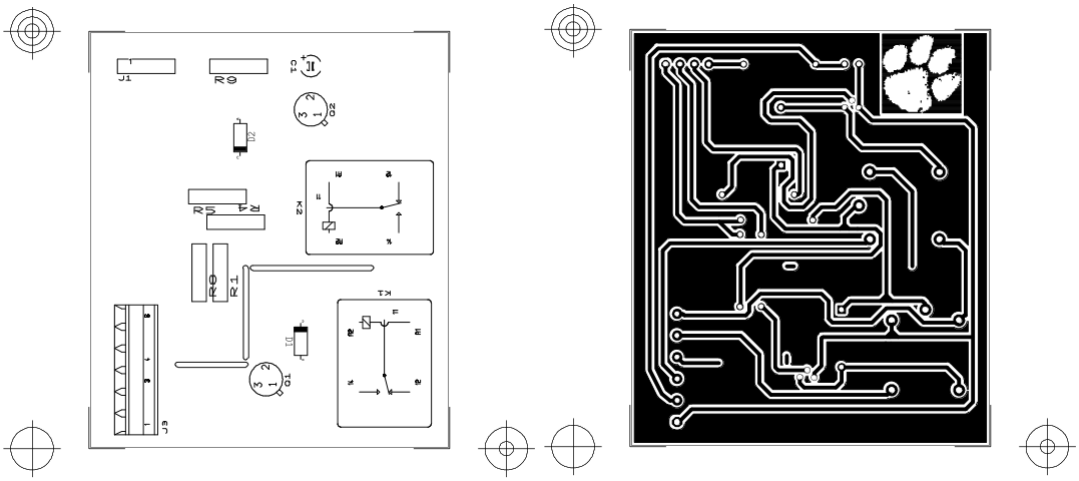


Figura 17: a) Circuito de control, vista superior. b) Circuito de control, vista inferior.

Diseño de la aplicación en LABVIEW

Para la aplicación de control y monitoreo se opto por utilizar el software LABVIEW de National Instruments. Este software es, en pocas palabras, una suite de instrumentación virtual. El proceso de comunicación es el siguiente:

- Abrir el puerto serial para comunicar con la tarjeta Arduino
- Si se desea enviar un comando de arranque, el programa enviara el carácter 'A' mediante el puerto serial.
- Si se desea enviar un comando de paro, el programa enviara el carácter 'P' mediante el puerto serial.
- Si se desea enviar un comando de cambio de giro, el programa enviara el carácter 'C' mediante el puerto serial.
- Si se desea enviar un comando de cambio de velocidad, el programa concatenara el carácter 'V' junto con el valor de un slider llamado 'Velocidad' en el panel de instrumentación, y enviara esta cadena mediante el puerto serial.

A continuación se detalla el diagrama de programación de la aplicación:

Figura 18: Panel de instrumentación de la aplicación en LABVIEW

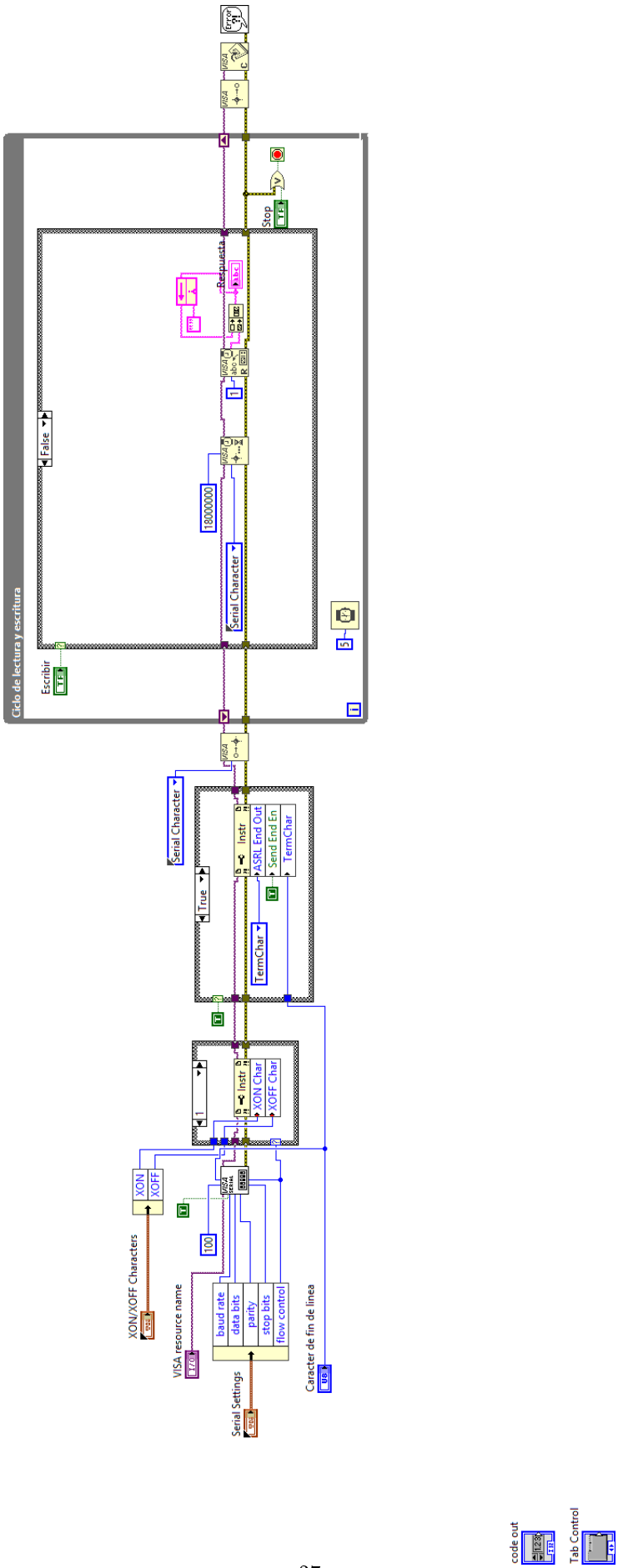


Figura 19: Bloque principal de código de la aplicación en LABVIEW

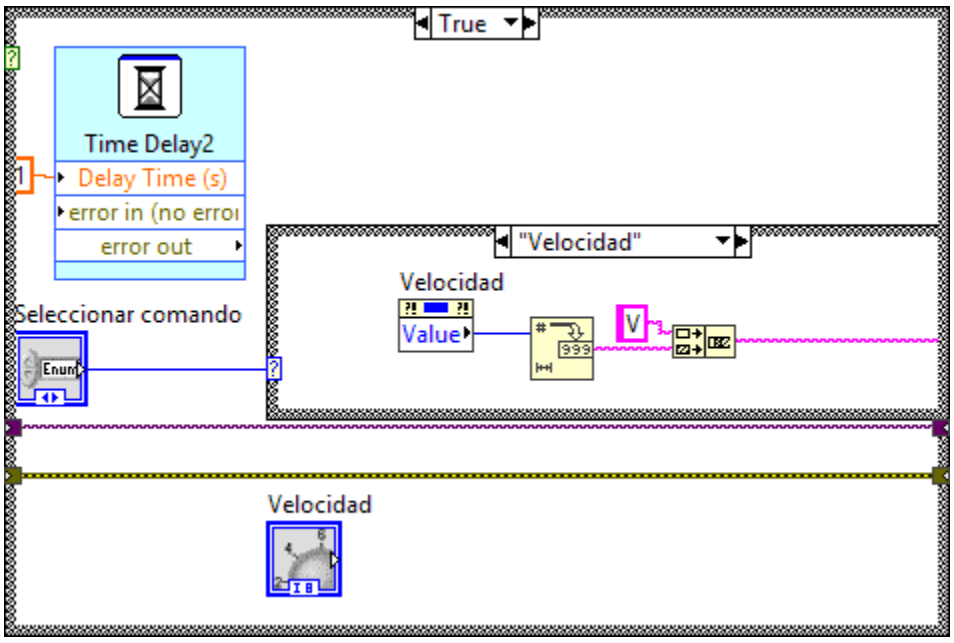


Figura 20: Condicional de envió de comando, al interior se puede observar el caso de comando 'Velocidad'

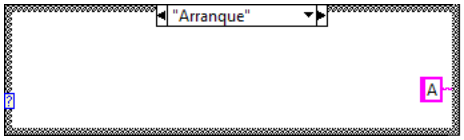


Figura 21: Condicional de comando de 'Arranque'

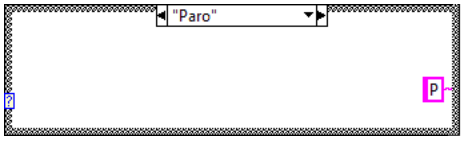


Figura 22: Condicional de comando de 'Paro'

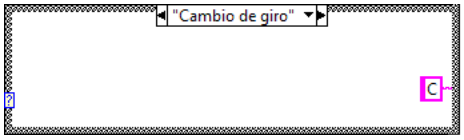


Figura 23: Condicional de comando de cambio de giro'

Utilizando la herramienta 'Application builder' de LABVIEW podemos exportar este programa a un archivo ejecutable, el cual puede correr desde cualquier computadora con sistema operativo Windows.

Conclusiones

En definitiva, la realización de este proyecto presento grandes retos, sobre todo en el área de microcontroladores. La diferencia entre las plataformas de aprendizaje y las de desarrollo profesional es muy grande; aun así, después de meses de estudio, se logro superar esa barrera entre los pequeños proyectos y las aplicaciones para producción. Es necesario notar que el protocolo CAN como tal es un poco difícil de implementar en un sistema *barebones* como lo es este, y requerirá algo de ingenio por parte del desarrollador para poder ser implementado de manera solida.

Es posible que adoptar algún protocolo como CANOpen o DeviceNet hubiera sido una mejor alternativa en consideración a la aplicación final, aunque estos no permitirían familiarizarse tanto con el funcionamiento en la capa física y de control de enlace como el protocolo CAN. Con el conocimiento adquirido durante la realización de este trabajo se pueden implementar proyectos tanto de instrumentación automotriz como de automatización industrial. Es necesario reconocer la flexibilidad que brinda un protocolo tan bien documentado y cimentado, que tentativamente, podríamos decir que es el de mayor difusión a nivel mundial hoy en día.

Bibliografía

[1] Renesas. *Introduction to CAN*. Renesas, 2006. URL http://elk.informatik.fh-augsburg.de/pub/rtlabor/ti-versuche/can/rej05b0804_m16cap.pdf.

[2] Zhen Yu. Control y monitorización de plantas mediante una red can. Master's thesis, 2011. URL http://upcommons.upc.edu/bitstream/handle/2099.1/12488/68403.pdf.

[3] NXP. Mpc5606b startertrak development kit. URL http://www.nxp.com/products/software-and-tools/hardware-development-tools/startertrak-development-boards/mpc5606b-startertrak-development-kit:TRK-MPC5606B? Ultimo acceso 20 de Julio de 2016.

[4] Microcontroller Solutions Group, 2012. URL http://cache.nxp.com/files/32bit/doc/ref_manual/MPC5607BRM.pdf. Ultimo acceso 13 de agosto de 2016.

[5] Arduino LLC. Arduino mega 2560. URL https://www.arduino.cc/en/Main/ArduinoBoardMega2560.

[6] Seeed Studio. Can bus shield. URL https://github.com/Seeed-Studio/CAN_BUS_Shield.

[7] Parker Hannifin Manufacturing Ltd. *690+ Series AC Drive Frame G, H & J Product Manual*, 5 edition, 2015. URL http://www.parker.com/Literature/SSD%20Drives/AC%20Drives/AC690/HA465084.pdf.

[8] CAN in Automation. History of can technology. URL http://www.can-cia.org/can-knowledge/can/can-history/. Ultimo acceso 18 de julio de 2016.

[9] Patrick Tounsi Alexandre Boyer. Presentation of mpc5604b (qorivva). 2013. URL http://alexandre-boyer.fr/alex/enseignement/Presentation_of_MPC5604B.pdf.