Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Master Universitario en Energías Renovables: Generación Eléctrica

E. Técnica Superior de Ingenieros Industriales y de Telecomunicación
Universidad Pública de Navarra (UPNA) Nafarroako Unibertsitate Publikoa

Proyecto Fin de Master:

"Diseño de equipo para configuración de dispositivos CANopen de aerogeneradores AW3000 de Acciona Wind power"

Autor:
Edgar Bueno Hernández

Pamplona, 27 de Septiembre de 2019

Resumen

CAN bus es un protocolo de comunicación para entornos distribuidos en el que varios dispositivos se comunican entre si, mediante la transmisión de mensajes en un bus común. Es un protocolo de comunicación robusto, con alta inmunidad al ruido eléctrico y mecanismos para verificar la integridad de los mensajes transmitidos. Es ampliamente utilizado en la industria y la de las renovables no es la excepción.

CANopen basado en CAN bus, es un protocolo avanzado de comunicación, estandarizado en la industria y que es utilizado en el sistema de control de ajuste de angulo de las palas, de los aerogeneradores AW3000 de Acciona Windpower.

El presente trabajo describe el diseño electrónico de un equipo de configuración de dispositivos CANopen, portátil y enfocado a las necesidades del técnico de mantenimiento.

Dedicado

A la memoria de mi Padre y de mi Madre....
Y a los que aun están conmigo, mi hermana Celeste
y mis queridos sobrinos Ethan y Fernanda.

Agradecimiento

Al Dr. Eugenio Gubía Villabona, por las recomendaciones para la defensa de este proyecto.

Índice

Índice de figuras.

Índice de tablas.

1. Introducción.

Entre las diferentes tecnologías de energías renovables la energía eólica, se ha consolidado como unas de las grandes industrias con mayor evolución tecnológica. Los aerogeneradores son cada vez de mayor potencia y ante las exigencias de eficiencia y seguridad de operación, los sistema de control se hacen cada vez más complejos, integrando en sus sistemas mayor número de sensores y actuadores. En consecuencia, el cableado aumenta y el flujo de señales de datos analógicas y digitales es cada vez mayor.

Para estructurar la conectividad y la comunicación entre PLCs, sensores y actuadores los aerogeneradores modernos implementan protocolos de comunicación industrial.

En el caso de los aerogenerador de Acciona Windpower AW3000 de 3MW se utiliza CANopen, un protocolo de comunicación de alto nivel basado en CAN BUS, que tiene como ventajas principales la alta inmunidad al ruido, la simplificación de la conectividad de diferentes dispositivos a través de un solo bus de comunicación común, entre otros. Por contraparte a nivel de diagnostico requiere de técnicos con mayor formación y equipos de diagnostico mas complejos.

En una red CAN OPEN, una de las tareas de los técnicos mas cotidianas previas al montaje de un sensor o de un actuador conectado a la red CAN, es su configuración. Que consiste en programar el dispositivo para asignar la velocidad de transmisión(baud rate), un numero de nodo que sera único en el bus CAN y programar la estructura de datos que serán enviados y recibidos.

Para configurar un dispositivo de la red CANopen de los aerogeneradores de Acciona, actualmente cuentan con un equipo que requiere de un ordenador. Por el alto precio del equipo, suele haber uno por parque eólico, por lo que el técnico de mantenimiento suele configurar los dispositivos en las oficinas de la subestación. El equipo al requerir de un ordenador tiene poca portabilidad y muy raramente es utilizado en el interior del aerogenerador donde un ordenador portátil es mas vulnerable a daños.

El equipo y el software utilizado para la configuración de los dispositivos CanOpen ha sido diseñado en exclusiva para estos aerogeneradores, sin embargo tiene una serie de desventajas, la principal la poca portabilidad, software poco intuitivo y equipo que requiere la intervención manual del operario en varias ocasiones durante el proceso de configuración.

El presente proyecto trata sobre diseño de un nuevo equipo para la configuración de sensores CANopen enfocado a los aerogeneradores de Acciona Windpower con los objetivos que se detallan a continuación.

1.1 Objetivos del proyecto.

El proyecto busca diseñar un equipo de configuración de dispositivos CANopen, enfocado a las necesidades del técnico de mantenimiento de los aerogeneradores de Acciona Wind power.

El equipo diseñado busca tener las siguientes características :

- Portabilidad.
- Interface de usuario intuitiva.
- Detección de baud rate y numero de nodo automática.
- Configuración del sensor con la mínima intervención del usuario.
- Detección automática del tipo de dispositivo para evitar cargar la configuración correspondiente a otro sensor.

Para el diseño del equipo se utiliza como plataforma de desarrollo Arduino, que son módulos de desarrollo de código abierto basado en los microcontroladores ATmega328p y Atmega2560 entre otros. Para posteriormente definidas las necesidades del proyecto diseñar un circuito impreso a medida.

Para el desarrollo del proyecto se utilizaran librerías de código abierto y librerías desarrolladas para este proyecto en el IDE de arduino.

El proyecto contempla, diseño electrónico, diseño mecánico, conocimiento del protocolo CANopen y conocimiento de las necesidades del técnico de mantenimiento en los aerogeneradores de Acciona Winpower.

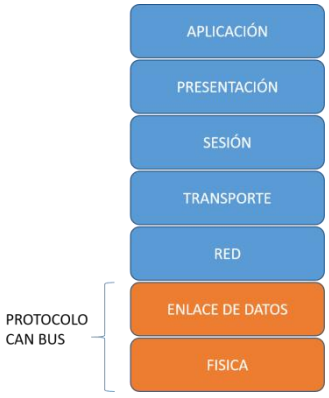
2. Protocolo CAN bus.

CAN bus(controlled Area Network) es un protocolo de comunicación creado a finales de los años 80 por la firma alemana Robert Bosh GmbH, fue inicialmente diseñado para la industria automotriz, para remplazar y simplificar el cada vez mas complejo cableado, por un bus de comunicación de dos hilos. Su ámbito de aplicación actual esta bastante extendido desde la industria del transporte, equipos industriales, equipos médicos, domótica, etc.

El protocolo esta diseñado para la transmisión de mensajes en entornos distribuidos, utiliza un bus común donde múltiples dispositivos(Nodos) son conectados.
Al compartir el bus de comunicación no hay forma de enviar un mensaje solo a un nodo especifico, todos los nodos compartirán invariablemente el trafico de la linea. Pero hay ciertos mecanismos en el protocolo que permiten que cada mensaje pueda ser filtrado por uno o mas nodos conectados a la red, ignorando el resto del trafico.

El protocolo CAN bus cubre las capas inferiores del modelo OSI(Open Systems Interconnection), las referentes a la capa física, relativas al medio en el que se lleva acabo la comunicación y la capa de enlace de datos que describe como se organizan los mensajes a través del bus.

Figura 2–1 Capa OSI de CAN BUS.



Las especificaciones relativas al enlace de datos y capa física del CAN Bus están definidas en la ISO 11898. Donde se define la comunicación transmitida entre dispositivos a través de la red.

2.1 Capa física de CAN bus.

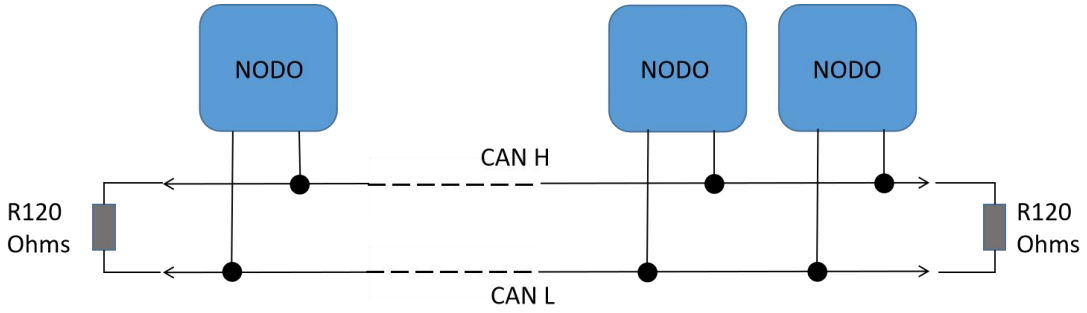
La ISO11898 define varios tipos de capas físicas, donde se especifican los niveles de señal, esquema de conexión del bus, impedancia del cable, velocidad de transmisión etc. La relativa a este proyecto es la definida en la ISO11898-2 conocida como CAN de alta velocidad, que describe una red CAN donde todos los dispositivos se conectan al mismo bus de comunicación, formado por dos lineas, denominadas CAN H(CAN HIGH) y CAN L (CAN LOW), donde en cada extremo del bus esta terminado

con una resistencia, para evitar reflexiones en la linea. Con esta configuración la velocidad máxima de transmisión del bus es de 1Mbit/s aunque puede verse limitada por longitud de la linea, el estándar recomienda par trenzado con apantallado, para mayor inmunidad al ruido. La siguiente tabla muestra las posibles velocidades de transmisión de acurdo a la longitud del linea.

Tabla 2-1 Baud Rate con respeto a lo longitud de la linea.

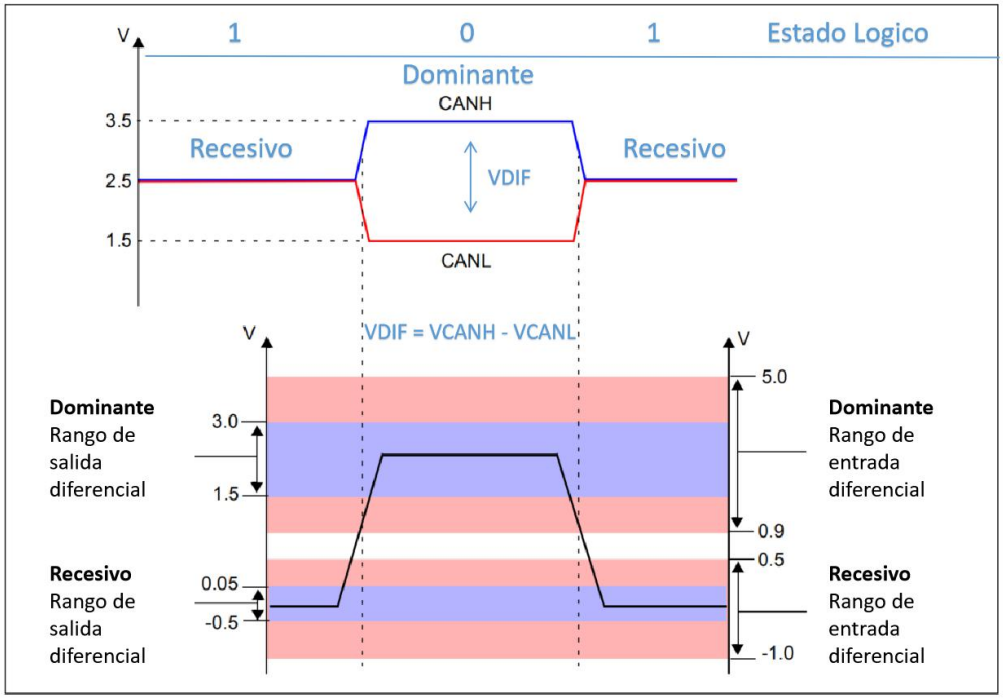
Baud Rate	Longitud Maxima	Resistencia terminal
1 Mbps	30 m	120 Ω
800 kbps	50 m	150-300 Ω
500 kbps	100 m	150-300 Ω
250 kbps	250 m	150-300 Ω
125 kbps	500 m	150-300 Ω
50 kbps	1000 m	150-300 Ω
20 kbps	2500 m	150-300 Ω
10 kbps	5000 m	150-300 Ω

Figura 2–2 Conectividad tipica de una red CAN



En la Fig. 2.2, se muestra la conexión típica de una red CAN, donde dos o mas nodos comparten el mismo canal de comunicación, las lineas CAN H y CAN L que son canales diferenciales, con dos estados, llamados Dominante y Recesivo, como lo muestra la siguiente figura 2.3 con sus valores de tensión y estado lógico.

Figura 2–3 Señal diferencial del CAN bus.



El estado Recesivo corresponde al 1 binario y el Dominante a 0. El canal diferencial es una de las características que desde el punto de vista de señal, hacen a CAN bus robusto y tolerante a ruidos. El estado Recesivo o Dominante no esta definido en función del nivel de tensión con respecto a GND, si no con respecto a la diferencia de tensión entre ambos canales VDIF = VCANH - VCANL.
Lo que permite que :

1. La comunicación pueda continuar aun en ausencia de una de las lineas.
2. Inmunidad al ruidos. Cualquier ruido eléctrico se induciría en ambas lineas y el diferencial de tensión VDIF se seguirá manteniendo.

Adicionalmente en la ISO11898-2 se especifica que los transductores de señal CAN de los diversos fabricantes deben sobrevivir a cortos circuitos desde -3V a +32V y soportar transitorios desde -150V a +100V. Dichas características desde el punto de vista de señal hacen a CAN bus un medio robusto de comunicación. Otras características adicionales a nivel lógico lo hacen aun mas robusto y se explican en las siguientes secciones.

2.2 CAN bus capa de enlace de datos.

El protocolo define que la comunicación entre nodos de una red CAN sera a través de cuatro tipos de tramas de comunicación (“Frames”):

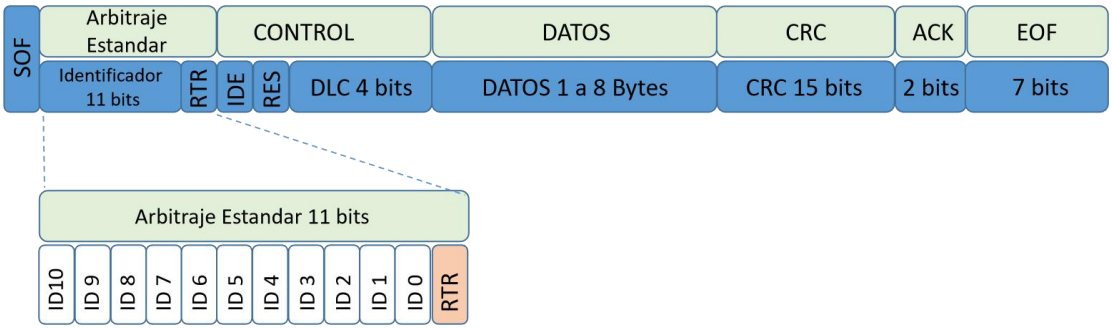
1. Trama de Datos (“Data frame”).
2. Tramas de petición de datos (“Remote Frame”).
3. Tramas de Error (Error frame”).
4. Trama de sobrecarga (“Overload Frame”).

2.2.1 Trama de Datos (Data Frame).

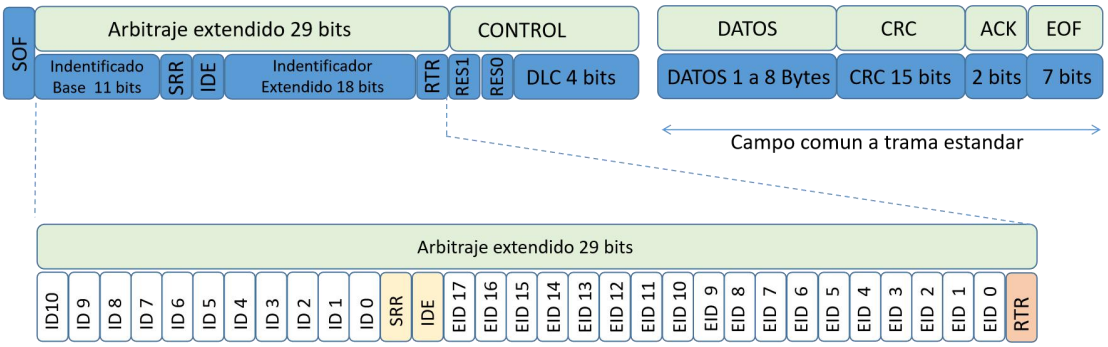
Es la trama utilizada para transmitir datos de hasta 8 bytes, la estructura de la trama puede ser “Trama estándar”(Standard frame) con identificador de 11 bits de la versión CAN2.0A o “trama extendida”(Extended frame) con identificador de 29 bits que fue incluida posteriormente en la versión de CAN2.0B, para aumentar el numero de identificadores posibles, manteniendo la compatibilidad con el formato anterior por lo que esta versión soporta ambos formatos de mensaje.
La figura 2.4 muestra la trama de datos para el formato estándar de 11 bits y el extendido de 29 bits.

Figura 2–4 Trama estándar y extendida.

Trama de datos Entandar de 11 bits



Trama de datos extendida 29 bits



Donde :

- SOF(Start of frame). Las tramas se inician con un bit dominante indicando el inicio de la transmisión de un mensaje y utilizado para la sincronización con el resto de los nodos.
- Arbitraje estándar: El siguiente campo es una identificador de mensaje, utilizado para el arbitraje, en caso de colisión, a menor valor del identificado, mayor prioridad en la transmisión del mensaje.
- RTR(Remote transmision request). Bit que indica que el mensaje es una petición remota si su estado es 1 o si su estado es 0 significa que es un mensaje de dato.
- IDE(Identifier extensión bit). Bit que indica si su valor es 0, que el mensaje es estandar, si su estado es 1, el mensaje es extendido y considerara los siguientes 18 bits como parte del identificador.

- RES(Reserved bit). Es un bit no utilizado, reservado por el estándar de CAN para posible uso futuro. Por defecto es un bit dominante (Estado 0). En la trama extendida son los bits RES1 y RES2 los bits reservados para uso futuro.
- DLC(Data lenght code). Es un campo de 4 bits que indica la longitud de bytes de datos incluidos en el mensaje de CAN.
- CRC(Cycle redundancy check). Campo utilizado para la detección de errores en los mensajes transmitidos. Su longitud es de 15 bits seguido de un bit delimitador recesivo (estado 1). Este campo es generado automáticamente por el controlador CAN del emisor y leído por todos los nodos en escucha para comprobar la integridad del mensaje .
- ACK (Acknowledge). Es el campo de aceptación o reconocimiento, formado por dos bits en el que uno de ellos es el bit ACK y el segundo un bit delimitador recesivo(estado 1). El bit de ACK es transmitido con estado recesivo, cuando el nodo receptor recibe el mensaje correctamente cambia el estado a dominante, detectando este cambio el transmisor, sabiendo que el mensaje ha sido recibido correctamente, en caso de que este bit permanezca en estado recesivo el transmisor asume que el menaje no ha sido recibido y volverá a transmitir el mensaje.

2.2.2 Trama de petición de datos("Remote frame").

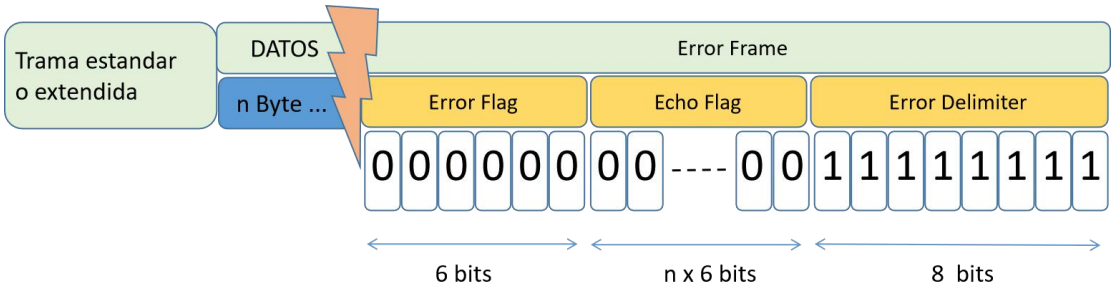
La trama de petición de datos, tiene la misma estructura que la trama de datos con dos diferencias importantes, el bit RTR(remote request transmision) es recesivo (estado lógico 0) indicando que el tipo de trama y no incluye el campo de datos por lo que los bits de control DLC (data long code) y datos es 0 indicando que la trama no contiene datos.

La trama remota tiene el propósito de solicitar datos con la trama del identificador que lleva el mensaje. La respuesta del receptor del mensaje no es automática solo indica al microcontrolador a través del controlador CAN, que un nodo solicita datos. Y el microcontrolador debe estar programado para transmitir los datos ante la recepción del mensaje.

2.2.3 Trama de error (Error Frame).

La trama de error, es una trama especial que viola la reglas de comunicación del CAN bus. El llamado bit stuffing que consiste que cuando hay en la trama mas de cinco bits del mismo valor se inserta un bit de valor inverso. En el caso de las trama de error, esta regla se incumple, la trama es transmitida cuando un nodo detecta un error en la comunicación interrumpiendo la transmisión en curso con 6 bits dominantes(Estado 0) en un campo denominado "error flag" que provoca que los demás nodos respondan a su vez con 6 bits dominantes, un campo denominado "echo error", finalmente la trama de error se cierra con un campo e 6 bits recesivos.
De esta forma el mensaje es destruido y detectado por el transmisor que volverá a retransmitir el mensaje automáticamente.

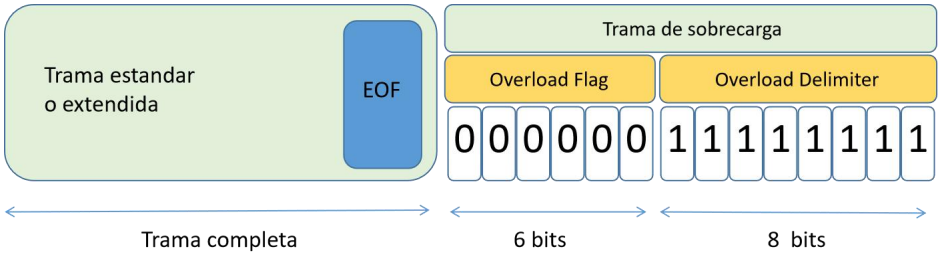
Figura 2–5 Trama de error.



2.2.4 Trama de sobrecarga.

Es una trama definida en el estándar que consiste en un campo de 6 bits dominantes(estado 0), seguido de un delimitador de 8 bits recesivo(estado 1), el mismo que el generado para una trama de error con la diferencia de que esta trama no interrumpe el mensaje en curso y es insertada al final de una trama de datos, cuando el controlador CAN del receptor no es capaz de procesar la recepción del mensaje por estar ocupado procesando una recepción anterior. Al enviar hasta dos posibles tramas de sobrecarga genera una pausa en el bus, esperando la retransmisión del siguiente mensaje. Actualmente la trama de sobrecarga en los controladores modernos aunque implementada, esta en desuso. Debido a la velocidad de procesamiento de los controladores CAN es difícil que una condición de sobrecarga pueda presentarse.

Figura 2–6 Trama de sobrecarga.



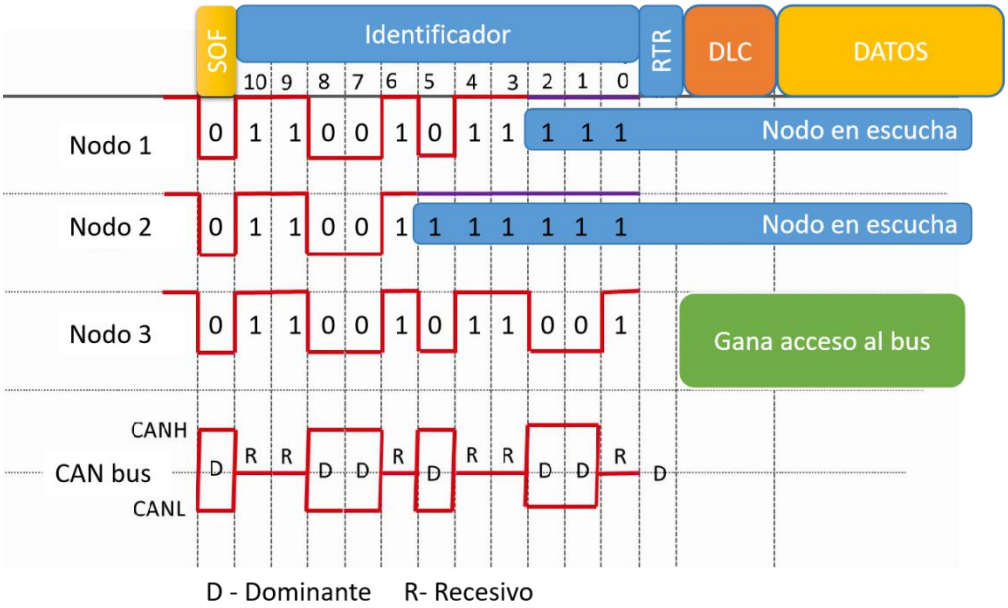
2.3 Control del acceso al medio(Arbitraje).

Al compartir todos los nodos el mismo bus de comunicación, CAN bus implementa ciertos mecanismos para organizar la transmisión de datos de cada nodo y evitar que los datos transmitidos se interfieran entre si. Para ello CAN bus implementa el algoritmo de “Acceso múltiple, con escucha de señal portadora, con detección de colisión y arbitraje sobre prioridad de mensaje”, mejor conocido por sus siglas en ingles como CSMA/CD+AMP (Carrier-Sense-Multiple Access, with collision detection and arbitration on message priority).

Cada nodo esperar un periodo de inactividad en el bus antes de intentar transmitir un mensaje. El orden de transmisión entre nodos es arbitrario, el primero que ocupa el bus, transmite. En caso de que dos nodos intenten transmitir al mismo tiempo, la

colisión en la transmisión es resuelta mediante un arbitraje a nivel de bits. El nodo que transmita un mensaje con identificador de mayor prioridad, gana el acceso al bus y los demás nodos se mantienen en escucha, a la espera de que se libere el bus para intentar transmitir un nuevo mensaje. El identificador de la trama estándar o extendida es el utilizado para tal fin, el identificador de mayor prioridad es aquel de menor valor en el identificador.

Figura 2–7 Ejemplo de arbitraje

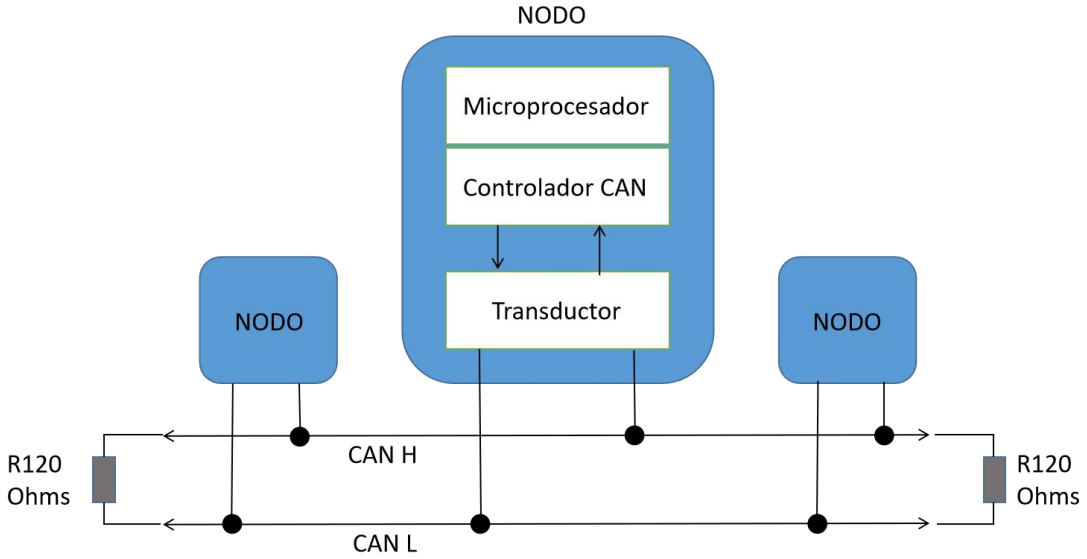


En el ejemplo hay 3 nodos transmitiendo al mismo tiempo en el CAN bus, en donde hay cierta coincidencia en los primeros bits del identificador. Los 3 nodos transmiten hasta el bit 7 donde el nodo 2 al estar en recesivo pierde el arbitraje a favor e los nodos 1 y 3 cuyo bit 7 es dominante. Los nodos 1 y 3 siguen transmitiendo al coincidir sus estados hasta el bit 2, donde el nodo 1 al ser recesivo pierde el arbitraje a favor del nodo 3 con bit dominante, ganando el acceso al bus para transmitir su mensaje.

2.4 Arquitectura del CAN bus.

La capa de datos y la capa física son normalmente transparentes al sistema de control. La capa física esta incorporada en circuitos electrónicos dedicados, que convierten la señal digital a la señal diferencial del CAN bus. Mientras que la capa de enlace de datos esta integrada en controladores de CAN bus. Que son circuitos electrónicos dedicados a procesar el trafico del CAN bus, apegados a los lineamientos del estándar.

Figura 2–8 Estructura de un nodo conectado al CAN bus



2.5 Mecanismo de detección de errores.

CAN bus es un protocolo de comunicación muy robusto debido a los diferentes mecanismos de verificación de errores. El protocolo incorpora cinco métodos de comprobación de errores, tres a nivel de mensaje de bits y dos a nivel de bit.
Si se detecta un fallo en la transmisión de un mensaje con cualquiera de estos métodos de detección de errores, el mensaje no se acepta y se destruye con una trama de error generada desde el receptor. Esto obliga al nodo transmisor a reenviar el mensaje hasta que sea recibido de forma correcta. Sin embargo si un nodo defectuoso cuelga un bus al repetir continuamente un error, el nodo es desconectado por el controlador después de alcanzar un limite de mensajes de error.

Los mecanismos de verificación de errores son los siguientes:

A nivel de mensaje:

1. **Comprobación de forma.** Esta comprobación busca campos en el mensaje que siempre deben ser bits recesivos. Si se detecta un bit dominante, se genera una trama de error. Los bits comprobados son el SOF(Start of frame), EOF(End of frame), Delimitador ACK y los bits delimitadores del CRC.
2. **Comprobación de errores CRC (Cycle redundacy check).** Cada trama de mensaje CAN lleva un campo de comprobación de errores, el CRC (cycle redundancy check) que es un checksum de todos los bits predecesores y que es calculado y transmitido como parte del mensaje por el transmisor y calculado mientras es transmitido por cada nodo de la red, si alguno detecta una discrepancia en el CRC, interrumpe el mensaje e inserta una trama de error, que al ser detectado por el transmisor hace que se reenvíe nuevamente el mensaje.
3. **Comprobación de aceptación del mensaje, bit ACK(Acknowleege check).** Cuando un mensaje va dirigido a un nodo determinado este bit transmitido en recesivo es conmutado a dominante por el receptor, indicando al transmisor que

el mensaje ha sido recibido. En caso de que este bit no cambie, significa que el mensaje no ha sido recibido y el transmisor lo reenviara de nuevo.

Comprobación a nivel de bit:

4. **Monitorización de bit.** Cada bit transmitido es monitorizado por el transmisor, si un bit se transmite en el bus y se lee el estado lógico opuesto al transmitido, el transmisor genera un error.
5. **Bits de relleno (Bit stuffing).** Cuando 5 bits consecutivos del mismo nivel son transmitidos por un nodo, el transmisor inserta un bit adicional con un nivel opuesto. Este bit extra es removido por el receptor. Esto se hace para asegurar que haya transiciones suficientes entre recesivo y dominante para garantizar la sincronización de los mensajes en todo el bus. Si mas de 5 bits del mismo nivel son detectados por algún nodo, se interrumpe el mensaje con una trama de error.

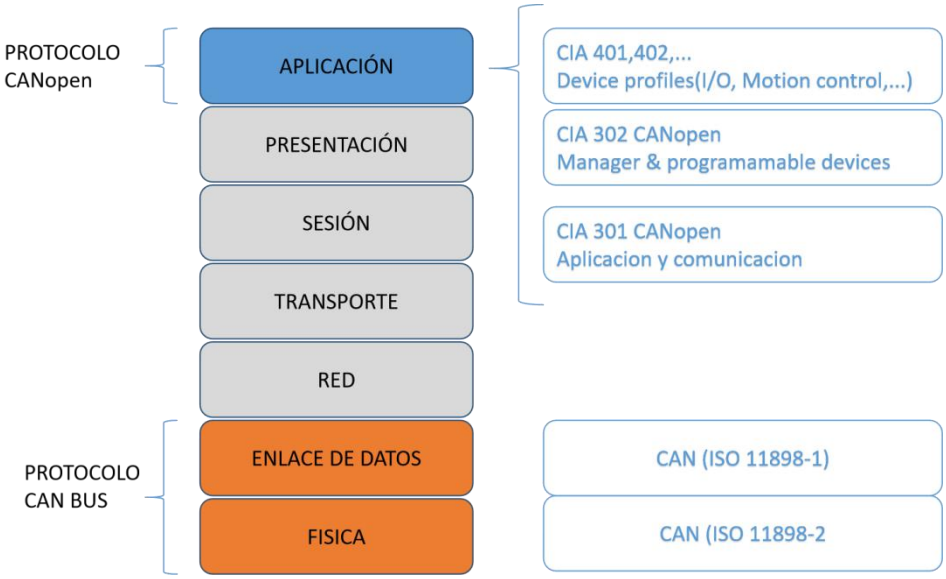
2.6 PROTOCOLO CANOPEN.

El protocolo CAN bus define la capa física y capa de enlace de datos. Definiciones insuficientes para construir una red CAN. Para que diversos dispositivos conectados puedan intercambiar información, se requiere ademas definir ciertas reglas que organicen como se lleva acabo ese intercambio, ¿Quien transmite?, ¿Como transmite?(identificador y datos transmitidos), ¿Cuando se transmite?, ¿Quien recibe información?, etc. Esta organización puede ser desarrollada a discreción por el diseñador de los dispositivos que se conectan a la red. Pero para que pueda haber comunicación entre dispositivos de diferentes fabricantes, se requiere estandarizar un modelo de comunicación común, en los que todos los dispositivos sigan las mismas reglas. Es decir se requiere de un protocolo de alto nivel común.

CANopen es uno de los protocolo de comunicación de alto nivel, basado en CAN bus, desarrollado por CIA(CAN in automation), asociación sin animo de lucro, formada por fabricantes y usuarios de CAN bus. CANopen define un protocolo que estandariza la comunicación entre dispositivos y aplicaciones de diferentes fabricantes.

En términos del modelo de sistemas de comunicación OSI (Open Systems Interconnection), el nivel de enlace de datos y la capa física son cubiertas por el protocolo CAN bus. CANopen cubre la capa de aplicación de modelo OSI.

Figura 2–9 Capas cubiertas del modelo OSI por el protocolo CANopen



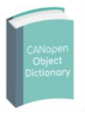
CANopen se pude explicar dividendo el protocolo en seis partes, que definen como se lleva acabo la comunicación en la red y como se organiza la información en los dispositivos conectados a ella. La siguiente figura resume el protocolo en sus componentes principales.

Figura 2–10 Partes de CANopen.

Modelo de comunicación: Definidos tres Modelos de comunicación entre dispositivos. Maestro/Esclavo, Cliente/Servidor y productor/consumidor.

Protocolo de comunicacion: Define protocolos usados para diferentes propositos de comunicacion. Ejemplo: Configuracion de nodos, transmision de datos, peticion de datos etc.

Estados de operación: Estados de operacion posibles en cada nodo, utilizando un protocolo especifico el nodo maestro puede cambiar el estado de los nodos esclavos.



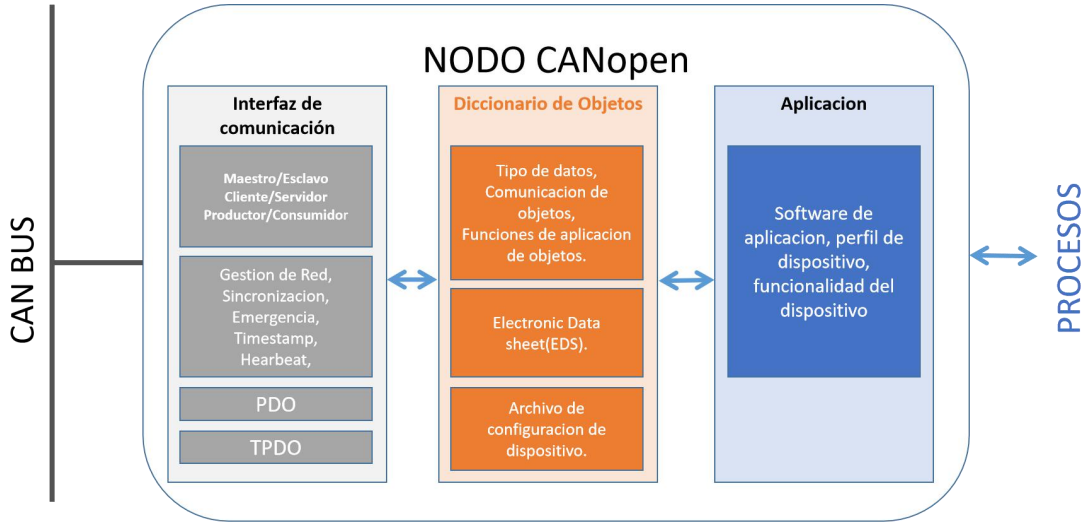
Diccionario de Objetos: Cada dispositivo tiene un diccionario de objetos, que es una tabla correspondiente a datos del dispositivo.

EDS(Electronic data sheet: Es un archivo estandarizado con extension *.INI o *.XML, que describe los campos del diccionario de objetos, permitiendo que a traves de herramientas de software actualizar datos de la tabla para configuracion.

Perfile de dispositivos estandarizados: El protocolo define los datos de entrada y salida de diversos dispositivos de acuerdo al campo de aplicacion e independientemente del fabricante.

Estas seis partes estructuran un dispositivo CANopen, la figura 2.11 muestra un esquema funcional de un nodo y como esta estructurado internamente.

Figura 2–11 Estructura de CANopen en un Nodo.



Donde la Interfaz de comunicación, lo componen todos aquellos proceso relacionados con la comunicación de los dispositivos con la red CAN. Los datos accesibles del dispositivo están contenidos en el diccionario de objetos y la parte de aplicación corresponde a los procesos propios del dispositivo trátese de sensores o actuadores, que interactúan a nivel global en la red CAN.
Las partes en las que se divide el protocolo se explican con mayor detalle a continuación.

2.6.1 Modelos de comunicación de CANopen.

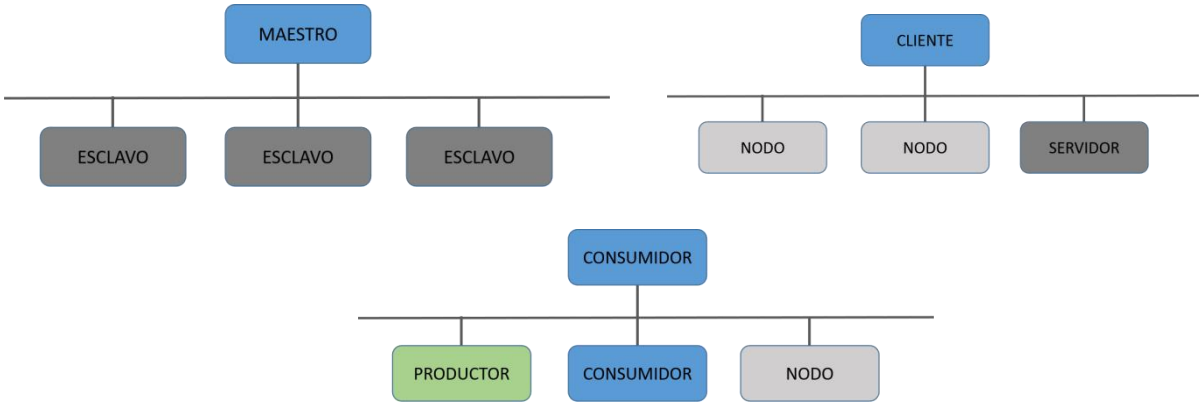
Los siguientes son los tres modelos de comunicación utilizados en el protocolo CANopen para la comunicación entre los nodos a través de la red. Son tres modelos relativamente similares utilizados con diferentes objetivos, que interactúan conjuntamente en el trafico de datos de un red CAN.

Maestro/Esclavo: En este modelo un nodo actúa como maestro, enviando o solicitando datos al resto de los nodos esclavos. El nodo maestro normalmente se trata de un PLC. Este modelo de comunicación es utilizado en el protocolo de gestión de red NMT (network managment). En donde el PLC maestro puede enviar ordenes para cambiar el modo de operación de los sensores.

Cliente/Servidor: Un nodo llamado cliente, envía una petición de datos a un nodo especifico que actúa como servidor, respondiendo con la información solicitada.
Esta información es cualquier dato contenido en el diccionario de objetos (OD).
Este modelo de comunicación es utilizado en el protocolo SDO(Service data object) que permite escribir o leer datos en el diccionario de objetos de cualquier nodo.

Productor/Consumidor: En este caso un nodo transmite datos periódicamente a la red CAN donde uno o mas nodos actúan como consumidores.

Figura 2–12 Modelos de comunicacion de CANopen.



2.6.2 Diccionario de Objetos CANopen.

Uno de los núcleos principales del protocolo CANopen es el diccionario de objetos, que es una estructura estandarizada que contiene todos los parámetros que describen el funcionamiento de un dispositivo CANopen y que permite su configuración mediante mensajes SDO (Service data object).
Los objetos están agrupados por funcionalidad e identificados con un índice de 16 bits y subíndice de 8 bits, si el objeto contiene mas de un elemento. El diccionario define el nombre de la variable, el tipo de variable, atributos de la variable(Permisos de Lectura-Escritura o solo lectura), entre otros datos. La figura 2.13 muestra un fragmento del diccionario de objetos de un dispositivo CAN.

Figura 2–13 Fragmento del diccionario de objetos CANopen

Index	Sub Ind.	Description	Index Type	Acc.	Initial Value	Annotation
1000H		**Device Type**	UNSIGNED32	ro	00020194 h	(DS 404 V1.2)
1001H		**Error Register**	UNSIGNED8	ro	00 h	ErrorRegister
1002H		**Manufacturer Status Register** (Calib. Date)	UNSIGNED32	ro	e.g. 20070712 (12. July 2007)	ManufacturerStatusRegister
1010H		**Store Parameters**				
	0H	largest subindex supported [3]	UNSIGNED8	ro	-	-
	1H	save all parameters	UNSIGNED32	rw	-	(Save all **Parameter**) ["evas"] [65 76 61 73]
	2H	save communication param.	UNSIGNED32	rw	-	(Save com. param. to **EEProm**) ["evas"]
	3H	save application parameters	UNSIGNED32	rw	-	(Save appl. param. to **EEProm**) ["evas"]

El diccionario de objetos se divide en secciones estandarizadas, en las que algunas entradas son obligatorias y otras son totalmente personalizables. Ver tabla.

Tabla 2-2 Índices del diccionario de objetos.

OD Index(16 bits, HEX)	Description
0000	Reserved
0001-025F	Data types
0260-0FFF	Reserved
1000-1FFF	Communication profile area
2000-5FFF	Manufacturer specific area
6000-9FFF	Device profile specific area
A000-BFFF	Innterface profile specific area
C000-FFFF	Reserved

La estructura del diccionario en todos los dispositivos, tiene como elemento común, el área de comunicación(communication profile) definidos en el estándar CANopen CIA301. En cambio el perfil de dispositivo(device profile area), depende del estándar aplicable para el tipo de dispositivo al que pertenezca, por ejemplo, CIA401 define el perfil de dispositivo(device profile) para equipos genéricos con señales de entradas y salidas, CIA406 para encoders, CIA408 para actuadores hidráulicos etc. El área especifica para el fabricante(Manufacturr specific area), da libertad al fabricante de asignar objetos que no estén definidos en los estándares existentes.

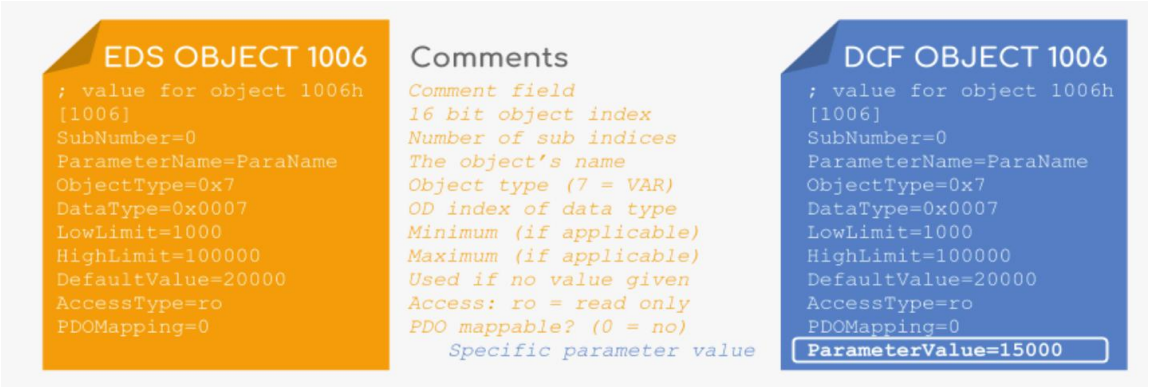
2.6.3 Electronic datasheet (EDS).

Previo a la utilización de cualquier dispositivo CANopen, requiere se configurado accediendo a los elementos del diccionario de objetos. Configurar manualmente cada uno de ellos resulta una tarea compleja por lo que programas informáticos específicos son utilizados para facilitar la tarea de configuración de dispositivos. El estándar CIA306, define un formato de archivo que describe el contenido del diccionario de objetos para ser utilizado por cualquier software de configuración de sensores CAN. El formato del archivo es básicamente de texto pero con extensión .INI y es proporcionado por el fabricante, donde el archivo contiene un listado de todos los objetos del diccionario, pero no sus valores de configuración.

Este archivo permite al usuario a través de un software generar un archivo adicional, copia del anterior pero con lo valores de configuración requeridos. Este archivo se conoce como DCF(Device configuration file).

La figura muestra un ejemplo del formato de los archivos EDS y DCF.

Figura 2–14 Formato EDS y DCF.

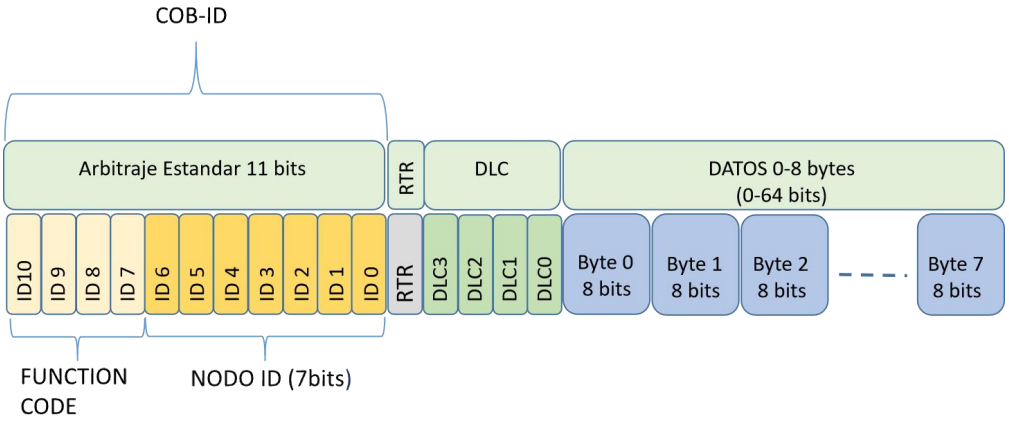


2.7 Protocolo de comunicación de CANopen.

Las tramas de comunicación CANopen adoptan la estructura del mensaje entandar. Los 11 bits del identificador se denomina COB-ID que esta dividido en dos partes, los 4 bits mas significativos para indicar el código de funciones (Function code) y los 7 bits restantes para identificación del numero de nodo(Node-ID). Al tener solo 7 bits para el identificador del nodo, el numero de nodos que soporta una red CANopen esta limitada a 128 nodos (2^7), con asignación de numero de nodo desde 0 a 127.

La figura siguiente muestra la estructura de un mensaje CANopen.

Figura 2–15 Trama de comunicación CANopen.



El COB-ID formado por Function code + Node-ID, permite identificar el tipo de objeto transmitido en el bus. Cada tipo de objeto cumple una función especifica en el protocolo. El listado de Objetos y de posibles valores de COB-ID se muestra en la siguiente tabla.

Tabla 2-3 Valores predefinidos de COB-ID de CANopen.

Description	Communication object	Function Code(bin)	Resulting COB-ID
Network Management	NMT	0000	0h
Syncronization	SYNC	0001	80h
Error codes	EMERGENCY	0001	81h-FFh (80h + Node-ID)
TIME STAMP	TIME	0010	100h
Transmit Process data object 1	TPDO1	0011	181h-1FFh (180h + Node-ID)
Receive Process data objetc 1	RPDO1	0100	201h-27F (200h + Node-ID)
Transmit Process data object 2	TPDO2	0101	28h-2FF (280h + Node-ID)
Receive Process data objetc 2	RPDO2	0110	301h-37Fh (300h + Node-ID)
Transmit Process data object 3	TPDO3	0111	381h-3FFh (380h + Node-ID)
Receive Process data objetc 3	RPDO3	1000	401h-47Fh (400h + Node-ID)
Transmit Process data object 4	TPDO4	1001	481h-4FFh (480h + Node-ID)
Receive Process data objetc 4	RPDO4	1010	501h-57Fh (500h + Node-ID)
Service data object Server-Client	SDO(Tx)	1011	581h-5FFh (580h + Node-ID)
Service data object Client-Server	SDO(Rx)	1100	601h-67Fh (600h + Node-ID)
Node guarding transmit error control	NMT Error Control	1110	701h-77Fh (700h + Node-ID)

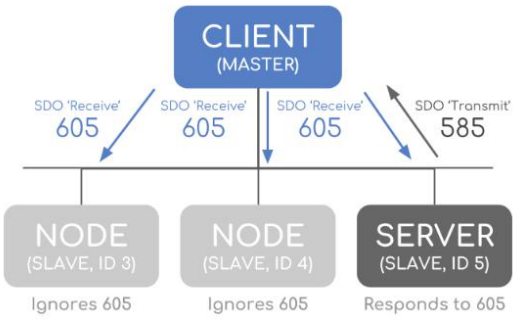
Siguiendo las reglas de arbitraje del protocolo CAN bus, la tabla anterior esta ordenada de acuerdo a la prioridad de acceso al bus. Así aquellos objetos relacionados con la administración de red NMT(Network management) con el COB-ID = 0h, son los objetos con mayor prioridad de transmisión en la red. Y los relativos a funciones de node guarding, los de menor prioridad por tener un valor de COB-ID mayor.

Los objetos de comunicación definidos en el protocolo son explicados a continuación.

2.7.1 Service Data Object (SDO).

CiA301 especifica varios servicios SDO. El propósito principal del protocolo es leer y escribir datos en una dirección del diccionario de objetos de un dispositivo CANopen. SDO sigue el modelo de comunicación Cliente-Servidor, donde el dispositivo poseedor del diccionario de objetos es el servidor y el dispositivo que accede al diccionario de objetos es el cliente, el cual es el único que puede hacer una solicitud de lectura o de escritura sobre un objeto del diccionario del servidor.

Figura 2–16 Protocolo SDO en CANopen.



Toda solicitud desde el cliente, viene seguida de una respuesta del servidor. Si la solicitud es de lectura, el servidor responde con el objeto solicitado. Si es de escritura, con la confirmación de que el objeto se ha escrito correctamente en el diccionario de objetos del servidor. Puede darse el caso que la solicitud no pueda ser atendida y el servidor responda con un mensaje de error con un código detallando el origen del fallo.

2.7.2 Formato del mensaje SDO.

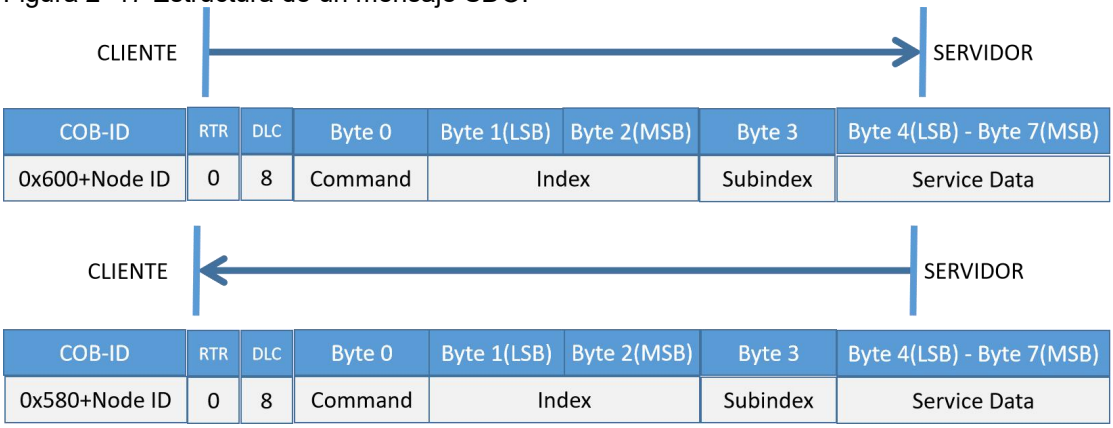
El protocolo SDO utiliza los COB-ID siguientes

Tabla 2-4 COB-ID del protocolo SDO.

COB-ID	Función
0x600+Node_ID	SDO Recibido (Rx)
0x580+Node_ID	SDO transmitido (Tx)

La dirección del mensaje Tx/Rx, están definidas en el protocolo desde el punto de vista del servidor. Desde un punto de vista practico, el Master(cliente) es quien siempre tiene la iniciativa y hace una petición al Esclavo(Servidor) enviando un mensaje con el identificador 0x600+Node_ID y el servidor responde con un mensaje con identificador 0x580+Node_ID.

Figura 2–17 Estructura de un mensaje SDO.



Los mensajes del protocolo SDO siempre tienen una longitud de mensaje de 8 bytes (DLC=8), aun cuando no sea necesario utilizarlos todos. Donde el byte 1 contiene un código de comando SDO que describe el tipo de operación a realizar (Lectura/escritura), el byte 1 y 2 indican el indice y el byte 3 el subíndice, que direccionan el objeto en el diccionario. Y los bytes 4-7 contienen los datos transferidos siguiendo siempre el formato "Little endian", el byte menos significativo primero.
El comando para el cliente esta formado siguiendo la siguiente estructura:

Tabla 2-5 Estructura del comando SDO.

Para el cliente:

Command Code bits	Meaning
7-5	CCS Client command specifier
4	Segment toggle bit
3-2	Data size n= 3(11b) 1 data byte sent n=2(10b) 2 data byte sent n=1(01b) 3 data byte sent n=0(00b) 4 data byte sent
1	expedited transfer
0	data set size is indicated

CCS value bits 7-5	Operation
0	Download segment request
1	Initiate download request
2	Initiate upload request
3	Upload segment request
4	not used
5	Block upload
6	Block download

Para el servidor:

Command Code bits	Meaning
7-5	SCS Server command specifier
4	Segment toggle bit
3-2	Data size n= 3(11b) 1 data byte sent n=2(10b) 2 data byte sent n=1(01b) 3 data byte sent n=0(00b) 4 data byte sent
1	expedited transfer
0	Data set size is indicated/End segment transfer(1)

SCS value bits 7-5	Operation
0	Upload segment response
1	download segment response
2	Initiate upload response
3	Initiatte upload segment response
4	Abort messagge
5	Block download
6	Block Upload

El listado de comandos simplificados para transferencia rápida de mensajes (expedited transfer) se listan a continuación.

Tabla 2-6 Listado de comandos del protocolo SDO.

Comando (HEX)	Tipo	Función
0x22	SDO(Rx), Solicitud de escritura(Download)	Escribe dato > 4 Bytes
0x23	SDO(Rx), Solicitud de escritura(Download)	Escribe dato 4 Bytes
0x2B	SDO(Rx), Solicitud de escritura(Download)	Escribe dato 2 Bytes
0x2F	SDO(Rx), Solicitud de escritura(Download)	Escribe dato 1 Byte
0x60	SDO(Tx), Confirmacion a cliente	Confirmación de escritura desde servidor
0x40	SDO(Rx), Solicitud de lectura(Upload)	Solicitud de paramaetro a servidor
0x43	SDO(Tx), Respuesta, envio de parametros.	Parametros a cliente. Unsigned 32 (4 bytes)
0x4B	SDO(Tx), Respuesta, envio de parametros	Parametros a cliente. Unsigned 16 (2 bytes)
0x4F	SDO(Tx), Respuesta, envio de parametros	Parametros a cliente. Unsigned 8 (1 byte)
0x80	SDO(Tx), Transferencia de datos abortada.	Envía mensaje de error a cliente.

2.7.3 SDO Expedited Read (Lectura rápida).

De la tabla anterior para una operación de Lectura(Upload) de un objeto en el diccionario con dirección 0x2001 que pertenece al área del perfil especifico del fabricante (Manufacturer specific profile area), donde esta registrado la velocidad de transmisión del dispositivo.

Figura 2–18 Ejemplo diccionario de objetos.

Manufacturer Specific Profile Area

Index	Sub Ind.	Description	Index Type	Acc.	Initial Value
2000H		Node-ID	UNSIGNED8	rw	1 d (Node-Id = 1)
2001H		Baudrate	UNSIGNED16	rw	20 d (20kbps)

Suponiendo que el dispositivo es el Nodo 5 (0x05), la comunicación se llevaría acabo de la siguiente forma:

Figura 2–19 Ejemplo comunicación SDO Expedited Read

CLIENTE → SERVIDOR

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x605	0	8	0x40	0x2001		0	0x00

CLIENTE ← SERVIDOR

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x585	0	8	0x4B	0x2001		0	0x00FA (250d)

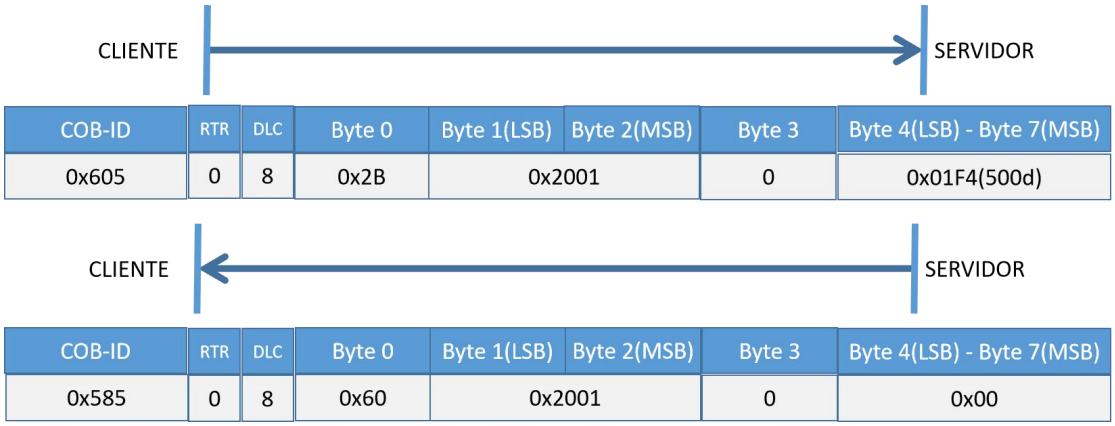
El cliente utiliza el COB-ID 0x600+Node_ID= 0x605, la operación solicitada es de lectura con el comando 0x40, para leer el contenido del diccionario de objetos con indice 0x2001 y subindice 0. Al ser petición de lectura no se envían datos.

El servidor responde con COB-ID 0x580+NodeID=0x585, indicando que el contenido es de una variable de 2 bytes(Unsigned16), confirmando el indice 0x2001, subíndice 0 y los bytes 4-7, contienen el calor del registro en este caso 0x00FA(250d), que corresponde al baud rate de 250Kbaud/s.

2.7.4 SDO Expedited Write(Escritura rápida).

Para una operación de escritura(Download) sobre el mismo objeto con indice 0x2001 del mismo dispositivo con Nodo 5, en el que se escribiría el baud rate 500kbits/s (0x1F4). La comunicación seria la siguiente:

Figura 2–20 Ejemplo comunicación SDO Expedited Write.



COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x605	0	8	0x2B	0x2001		0	0x01F4(500d)

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x585	0	8	0x60	0x2001		0	0x00

El cliente utiliza el COB-ID 0x600+Node_ID= 0x605, la operación solicitada es de escritura para una variable de 2 bytes(Unsigned 16), el comando es 0x2B, para escribir sobre el objeto del diccionario con indice 0x2001 y subíndice 0. El dato a escribir es 0x01F4 (500d) contenido en los bytes 4 al 7.

El servidor responde con COB-ID 0x580+NodeID=0x585, confirmando que el valor ha sido escrito correctamente (Comando 0x60), sobre el objeto con indice 0x2001, subíndice 0. Los bytes 4-7 no contendrán datos.

2.7.5 SDO Read Segmented (Lectura por segmentos).

La capacidad máxima de transferencia de datos en un solo mensaje SDO, es de solo 4 bytes, suficiente para la mayoría de Objetos. Sin embargo existen objetos de mas de 4 bytes, para lo cuales el protocolo SDO tiene un mecanismo llamado transferencia(Lectura/Escritura) por segmentos. El protocolo se explica a continuación.

El cliente solicita lectura de un objeto al servidor tal como se hizo en el expedited Read(lectura rápida), con el formato de comunicación siguiente:

Figura 2–21 Protocolo SDO Read Segmented.

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x600+Node_ID	0	8	0x40	Index		Sunindex	

La solicitud de lectura utiliza el comando 0x40 anteriormente utilizado. La diferencia esta en la respuesta del servidor que no puede transferir la información contenida en un solo bloque por superar los 4 bytes. Por lo que informa al cliente que el contenido del objeto solicitado supera los 4 bytes y que requiere lectura por segmentos, indicando al cliente en el campo de datos la longitud total del objeto a transmitir. La respuesta desde el servidor seria:

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x580+Node_ID	0	8	0x41	Index		Subindex	Data Lenght

Donde el comando 0x41(b1000001) viene de :

Command Code bits	Value	Meaning
7-5	010b	SCS Serrver command specifier (initiate upload response 010b)
4	0	Not used
3-2	0	Not used
1	0	Not used
0	1	Data set size indicated

El cliente deberá leer los datos por segmentos, con el formato de mensaje siguiente:

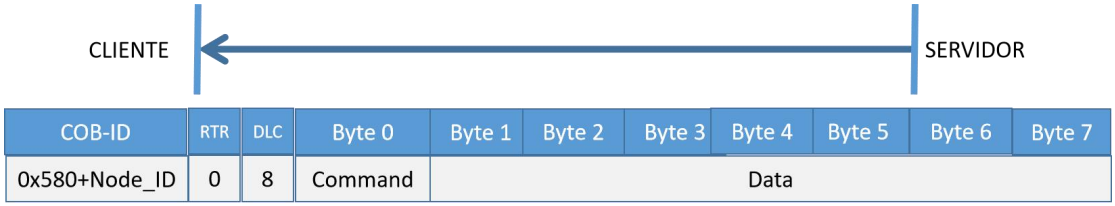
COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x600+Node_ID	0	8	0x60/0x70	Index		Subindex	

Comando 0x60 (b1100000) y 0x70(b1110000):

Command Code bits	Value	Meaning
7-5	011b	CCS Client command specifier (Upload segment request 011b)
4	0b/1b	toggled bit, must be flipped each request(start with 0)
3-2	00b	Not used
1	0b	Not used
0	0b	Not used

El mensaje se envía por segmento solicitado, donde el toggle bit del formato del comando debe conmutar su valor en cada una de las solicitudes, empezando la solicitud del primer segmento con 0, de allí el comando 0x60 (toggle bit 0) y 0x70 (toggle bit 1)

El cliente por su parte devuelve parcialmente el objeto en cada solicitud de segmento, con el siguiente formato:



COB-ID	RTR	DLC	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x580+Node_ID	0	8	Command	Data						

Command:

Command Code bits	Value	Meaning
7-5	000b	SCS Client command specifier (Upload segment request 011b)
4	0b/1b	toggled bit, must be flipped each request(start with 0)
3-1	000b	Not used
0	0b	Not used

Para entender mejor el protocolo considérese el Nodo 5. Partiendo de un fragmento de diccionario de objetos, el objeto con indice 0x1008 “Vendor Name” es un objeto de tipo String de solo lectura, con el contenido Atos S.p.a. Esto almacenado en formato digital corresponde a los valores ASCII de los caracteres, 0x41, 0x74, 0x6F, 0x20, 0x53, 0x2E, 0x70, 0x2E, 0x61, dato de longitud de 9 bytes.

Figura 2–22 Fragmento del diccionario de objetos.

Manufacturer Info								
1008h	00h	Vendor Name	Manufacturer device name	CONST			STR	Atos S.p.a.
1009h	00h	Driver Hardware Version	Manufacturer hardware version	CONST	Lev 1	Pag 32	STR	
100Ah	00h	Driver Software Version	Manufacturer software version	CONST	Lev 1	Pag 32	STR	

La comunicación iniciaría con el siguiente intercambio de mensajes:
El cliente solicita Lectura del objeto con indice x1008:

Figura 2–23 Ejemplo SDO Read Segmented.

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x605	0	8	0x40	0x1008		0x00	0x00

El servidor responde:

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x585	0	8	0x41	x1008		x00	x09 (Data Lenght)

El cliente solicita lectura del objeto x1008. El servidor informa al cliente mediante el comando 0x41 que el contenido el objeto solicitado supera los 4 bytes y requiere lectura por segmentos, indicando al ciente en el campo de datos la longitud total del objeto a transmitir, en este caso 9 bytes. El cliente tendrá que solicitar el dato en dos segmentos.

Primer segmento:

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x605	0	8	0x60	0x1008		0x00	0x00

COB-ID	RTR	DLC	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x585	0	8	0x00	0x41, 0x74, 0x6F, 0x20, 0x53, 0x2E, 0x70						

Segundo segmento y fin de transferencia:

COB-ID	RTR	DLC	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x605	0	8	0x70	0x1008		0x00	0x00

COB-ID	RTR	DLC	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x585	0	8	0x11	0x2E, 0x61						

SDO write segments(escritura por segmentos) es una operación similar a la lectura pero con comandos diferentes tanto para cliente y servidor. Puesto que esta operación aunque esta contemplada en el entandar es muy rara vez utilizada, no se explica ni se utiliza en este trabajo.
Existe también un protocolo SDO llamado transferencia por bloques, que es un método alternativo a la transferencia por segmentos, simplificado pero no soportado por todos los dispositivos, por lo que solo se hace esta breve referencia a el.
Para mayores detalles del protocolo SDO referirse al estándar CIA301.

2.7.6 SDO Abort code.

Cuando ocurre algún error de comunicación, el protocolo SDO contempla, mensajes específicos para abortar la operación, estos pueden darse por diversas causas, entre ellos Intento de lectura o escritura sobre un registro inexistente, intento de escritura sobre un registro de solo lectura etc. El formato del mensaje y el listado de códigos de error posibles se muestran a continuación.

Figura 2–24 Formato SDO Abort Code.

COB-ID	Byte 0	Byte 1(LSB)	Byte 2(MSB)	Byte 3	Byte 4(LSB) - Byte 7(MSB)
0x580+Node ID	x80	Index		Subindex	Abort code

El mensaje SDO abort code normalmente lo genera el servidor cuando la petición del cliente no puede ser atendida, sea petición de lectura o de escritura. El listado de códigos de error(abort code) se lista a continuación:

Tabla 2-7 Códigos de error SDO.

Abort code	Description
0503 0000h	Toggle bit not alternated.
0504 0000h	SDO protocol timed out.
0504 0001h	Client/server command specifier not valid or unknown.
0504 0002h	Invalid block size (block mode only).
0504 0003h	Invalid sequence number (block mode only).
0504 0004h	CRC error (block mode only).
0504 0005h	Out of memory.
0601 0000h	Unsupported access to an object.
0601 0001h	Attempt to read a write only object.
0601 0002h	Attempt to write a read only object.
0602 0000h	Object does not exist in the object dictionary.
0604 0041h	Object cannot be mapped to the PDO.
0604 0042h	The number and length of the objects to be mapped would exceed PDO length.
0604 0043h	General parameter incompatibility reason.
0604 0047h	General internal incompatibility in the device.
0606 0000h	Access failed due to a hardware error.
0607 0010h	Data type does not match; length of service parameter does not match.
0607 0012h	Data type does not match; length of service parameter too high.
0607 0013h	Data type does not match; length of service parameter too low.
0609 0011h	Sub-index does not exist.
0609 0030h	Value range of parameter exceeded (only for write access).
0609 0031h	Value of parameter written too high.
0609 0032h	Value of parameter written too low.
0609 0036h	Maximum value is less than minimum value.
0800 0000h	General error.
0800 0020h	Data cannot be transferred or stored to the application.
0800 0021h	Data cannot be transferred or stored to the application because of local control.
0800 0022h	Data cannot be transferred or stored to the application because of the present device state.
0800 0023h	Object dictionary dynamic generation fails or no object dictionary is present (e.g. object dictionary is generated from file and generation fails because of a file error).

2.8 Process Data object (PDO).

Para el intercambio de datos en una red CANopen, seria posible utilizar el protocolo SDO para leer objetos de cualquier dispositivo (Expedited read). Con la limitación de que el protocolo puede transmitir un solo objeto de 4 bytes máximo. Y por cada objeto solicitado, se requieren dos mensajes, petición desde el cliente y respuesta

desde el servidor. En una red formada por varios nodos implica un aumento del trafico, una transferencia de información lenta y un control de procesos complejo.

CANopen utiliza otros métodos de comunicación mas eficientes para el intercambio de datos en tiempo real, los PDO(process data objects), protocolo utilizado para transmitir rápidamente información de proceso en una red CANopen. Los PDO, utilizan el modelo de comunicación Productor-consumidor. Estos mensajes tienen capacidad de transmitir hasta 8 bytes de datos de proceso, enviados en una sola trama de comunicación CAN utilizando un COB-ID especifico para cada tipo de PDO.

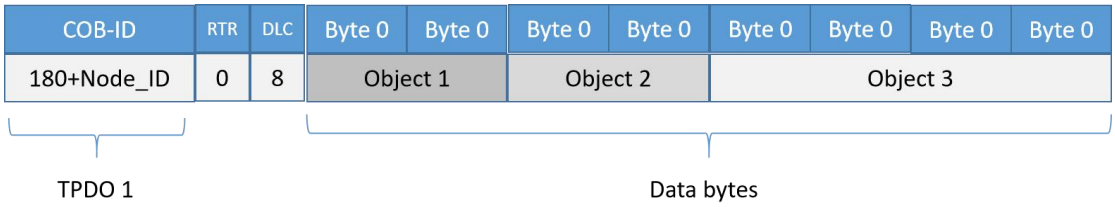
Hay dos tipos de PDOs:

- **TPDO(Transmit process data object):** Estos mensajes son utilizados para transmitir datos de proceso de un dispositivo hacia el control. El tipo de datos transmitidos pueden ser valores de un sensor, estado del dispositivo, etc. El estandar degfine la posibilidad de configurar hasta 4 TPDOs.

- **RPDO(receive process data object):** Son mensajes que recibe el dispositivo desde el control con consignas, como cambio de estado, de operación, consigna de control, etc. Hasta 4 RPDO pueden ser configurados.

Los datos transmitidos en un TPDO y en un RPDO, son totalmente personalizables durante la configuración inicial de un dispositivo CANopen. La capacidad de 8 bytes en el campo de datos puede ser utilizado para transmitir mas de un solo objeto.

Figura 2–25 Estructura de un TPDO.



Tanto el productor como el consumidor deben saber como esta estructurado el mensaje, para una vez recibido, extraer los objetos transmitidos. En el caso de los RPDO el contenido y el orden de los objetos se define en la configuración inicial del sensor.

La transmisión de los PDOS puede darse de las siguientes formas:

- Síncrona(Synchrona).
- Asíncrona por evento o tiempo (event driven asynchronus or time driven).
- A petición del consumidor(asynchronus requested by consumer).

La transmisión asíncrona por evento, significa que el nodo ha sido configurado para transmitir el TPDO, tras la ocurrencia de un evento, un cambio de la medida de un sensor, si la medida sobrepasa ciertos limites, etc. O si ha sido configurado como “time driven”, significa que el sensor transmitirá el TPDO periódicamente en un tiempo configurado. Los eventos que producen la transmisión del TPDO, son

configurados en los parámetros de comunicación del diccionario de objetos del dispositivo.

La transmisión a petición del consumidor, es un modo de transmisión de TPDO, que responde a los mensajes del consumidor que envía un mensaje Remote request, que consiste en un mensaje con el COB-ID del TPDO del nodo al que se solicita el datos con el bit RTR =1, DLC y campo de datos =0. En la practica no es un método recomendado para la transmisión periódica de datos.

En la transmisión Síncrona, los TPDOs de todos los nodos son transmitidos al Bus Can, tras recibir un mensaje desde el control llamado "Sync telegram". El formato del mensaje Sync tiene el COB-D: 0x80 y no lleva datos.

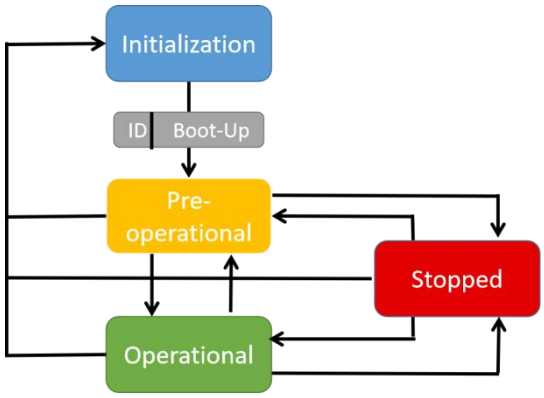
Figura 2–26 Mensaje de sincronizacion SYNC.

COB-ID	RTR	DLC	Byte 0	Byte 0	Byte 0	Byte 0	Byte 0	Byte 0	Byte 0	Byte 0
0x80	0	0	0x00							

2.9 Network management (NMT).

A nivel de comunicación los dispositivos diseñados para CANopen, funcionan como una maquina de estados. Donde cada estado define el comportamiento del dispositivo conectado a la red. Los estados posibles son los mostrados en la figura Siguiente.

Figura 2–27 Maquina de estados de un dispositivo CANopen.



Donde:

1. **Initializing:** Es el estado inicial al que entra el dispositivo después del encendido o tras un reset. Es este estado el nodo esta inactivo a nivel de comunicación.

Se cargan las configuración del dispositivo definida en el diccionario de objetos y los parámetros de comunicación (Numero de Nodo, bau rate, etc). Una vez finalizado estos procesos el nodo transmite un mensaje de Boot-up con el que informa a los demás dispositivo que se encuentra conectado y pasa automáticamente al estado al

estado Pre-Operational o directamente al estado operacional si el dispositivo esta configurado para ello.

Figura 2–28 Formato del mensaje Boot Up.

COB-ID	RTR	DLC	Byte 0
0x700+NodeID	0	0	0

2. **Pre-operational:** En este estado, el dispositivo habilita la comunicación, pero la transmisión automática de los PDO (Proceess data object) esta deshabilitada y solo responde a comandos NMT, SDO, SYNC, Time stamp o Hearbeat si esta habilitado. Los servicios SDO(Service data object) pueden ser utilizados par configurar el dispositivo, el cual es solo posible es este estado e operación. Mediante un comando NMT, el dispositivo puede pasar al siguiente estado de operación.

3. **Operational state:** En este estado el dispositivo esta activamente operativo y todos los objetos de comunicación habilitados.

4. **Stopped**: Este estado deja inactivos todos los objetos de comunicación posibles, excepto a los comandos NMT que pueden cambiar a cualquiera de lo estados restantes y Heartbeat.

El comando NMT esta basado en el modelo de comunicación Maestro-Esclavo. El dispositivo Maestro controla vía protocolo NMT el estado de operación de los dispositivos conectados a la red. El formato de los mensajes NMT es el siguiente:

Figura 2–29 Comando NMT.

			NMT Function	Target Node
COB-ID	RTR	DLC	Byte 0	Byte 1
0x0000	0	2	0xNN	0xNNNN

Donde “Target Node” puede ser cualquiera de los 127 Nodos posibles (0x01-0x7F) o 0x00 para que la orden sea dirigida a todos los nodos. Y “NMT function” puede ser cualquiera de los siguientes valores.

Tabla 2-8 Comandos NMT.

Code	NMT Function
0x01	Enter Operational
0x02	Enter STOP
0x80	Enter Pre-Operational
0x81	Reset Node
0x82	Reset Communication

Ademas de los cambios de estado, el comando NMT incluye dos tipos de reset, ambos Reset llevan al sensor al estado de inicialización:

Reset Node o Reset application: Donde los parámetros del diccionario de objetos específicos del fabricante son cargados nuevamente a la configuración del dispositivo.

Reset Communication: Donde el dispositivo, desde el diccionario de objetos actualiza únicamente la configuración relativa a la comunicación (Numero de Nodo, baud rate).

Estos comandos son utilizados cuando la configuración del sensor ha sido modificada en el diccionario de objetos. Si solo los campos relativos a la comunicación han sido modificados, el "Reset communication" actualiza el dispositivo con la nueva configuración. Si otros elementos del diccionario de objetos han sido modificados, entonces el Reset Node es requerido para actualizar la configuración del dispositivo.
Con esta ultima parte se concluye el resume del Protocolo CANopen. El protocolo es complejo y bastante extenso, se han explicado los componentes del protocolo mas relevantes para el proyecto. Para mayores referencias consultar CIA301.

2.10 Ventajas del protocolo CAN bus y CANopen.

Descritas las características del protocolo CAN bus y CANopen, se resumen su ventajas.

- Protocolo de comunicación distribuido que reduce el cableado al utilizar dos lineas para interconectar varios dispositivos en el mismo bus de comunicación.
- El canal diferencial utilizado en el CAN bus(CAN H, CANL), tiene por sus características alta inmunidad al ruido, el par trenzado y el cable apantallado aumenta aun mas sus características de inmunidad.
- Cuenta con algoritmos para la detección de errores de transmisión lo que favorece la fiabilidad de la comunicación a lo largo del bus.
- Retransmisión automática ante detección de errores en el bus.
- Algoritmo CSMA/CD+AMP que organiza la comunicación en el bus mediante un sistema de arbitraje que evita colisiones entre las diferentes tramas transmitidas, evitando que el bus se congestione.
- Velocidad de transmisión de hasta 1MBS.
- Bajo costo de los controladores CAN y transductores.
- Protocolo diseñado para aplicaciones en tiempo real.
- Soporte de perfiles de dispositivos.

Por otro lado algunas de sus desventajas es el numero limitado de datos que puede transmitir 8 bytes máximos. No es un protocolo apropiado para la transmisión de datos de vídeo, audio, imagen etc. Pero para la comunicación entre actuadores y sensores que transmiten datos limitados es un protocolo muy potente.

3. Desarrollo del Proyecto.

Para cumplir con los objetivos del proyecto, el desarrollo se ha dividido en cuatro partes. Primero con el análisis de los dispositivos CANopen para los que el equipo estará diseñado. Definiendo la configuración que estos dispositivos deberán tener. La segunda parte es el desarrollo de los algoritmos con el protocolo CANopen, para configurar los dispositivos CANopen del los aerogeneradores AW3000. Estos algoritmos deberán integrarse en el programa del microcontrolador. La tercera es el desarrollo del Hardware que incluye conceptualización del producto y diseño electrónico y mecánico. Y por ultimo el programa del microcontrolador que da vida al Hardware del equipo.

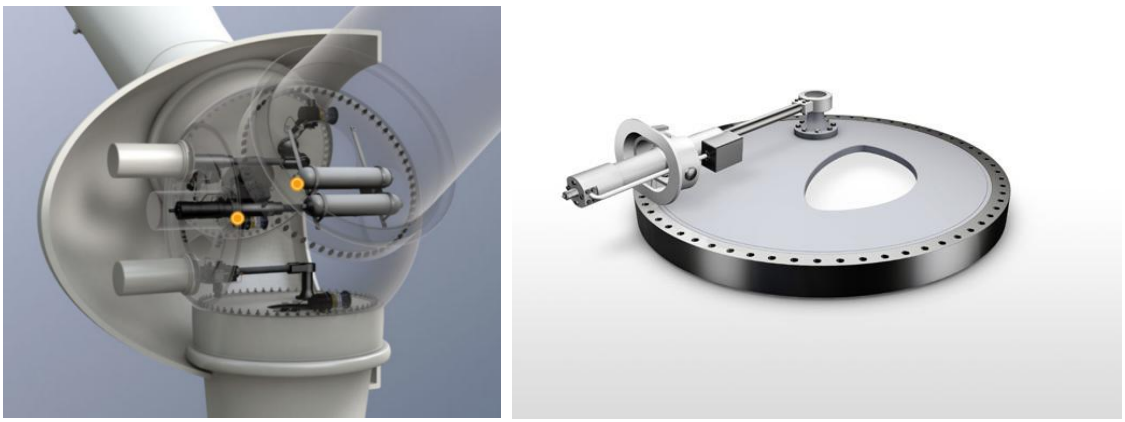
3.1 CANopen en los aerogeneradores de Acciona windpower.

Una parte muy importante del sistema de control de aerogeneradores, es el control del ajuste del angulo de palas, conocido como “pitch control”. Cuya función es maximizar la potencia del aerogenerador, controlar las revoluciones del rotor y frenar de forma aerodinámica el rotor del aerogenerador.
El control de pitch ajusta el ángulo de las palas del rotor para maximizar la potencia del aerogenerador ante variaciones de la velocidad del viento. Cuando la velocidad del viento supera la potencia máxima permisible del generador, el pitch ajusta el angulo de las palas para reducir la eficiencia aerodinámica y mantener la velocidad del rotor constante. Para el frenado aerodinámico el control de pitch ajusta el angulo de las palas a 90^{o}, lo que produce una disminución rápida de las revoluciones del rotor, sin el estrés mecánico que supone la activación del freno de disco.

Para el control del pitch se utilizan mayormente dos sistemas, el eléctrico, en el que cada pala va unida a una corona acoplada a un motor con reductora. Y el sistema hidráulico que utiliza pistones hidráulicos para mover el angulo de palas.

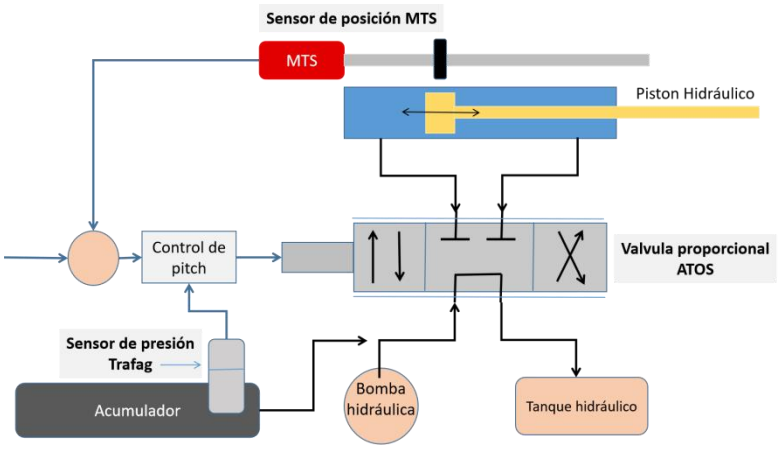
En el caso de los aerogeneradores de acciona windpower de 3MW modelo AW3000, utilizan el sistema hidráulico para el control del pitch.

Figura 3–1 Sistema de control de Pitch en aerogeneradores.



Un esquema simplificado de los elementos que conforman el sistema de control de pitch para cada pala, se muestra e la figura 3.2. Una válvula proporcional (actuador ATOS) es activada por el control, para extender o contraer el pistón hidráulico hasta la posición de consigna cuya retroalimentación depende del sensor de posición (sensor MTS). Los sistemas de control de pitch tiene como sistema de seguridad acumuladores hidráulicos, que en caso de una emergencia o perdida total de energía, la presión acumulada es utilizad para mover la pala a través de pistón hidráulico, a la posición de seguridad (posición de bandera 90 grado). La presión del acumulador es critica para la seguridad del sistema y es medida por un sensor de presión (sensor Trafag).

Figura 3–2 Esquema sistema hidraulico de una pala.



Loas aerogeneradores de Acciona Windpower AW3000 utilizan CANopen para el control del pitch. En donde el PLC de la nacell esta conectado por la junta rotativa, a la red CAN que comunica con los sensores y actuadores del sistema hidráulico del pitch. El listado de componentes CANopen por Pala son los siguientes:

- Válvula proporcional ATOS.
- Sensor de posición MTS Temposonic
- Sensor de presión TRAFAG.

3.1.1 Válvula proporcional ATOS.

La válvula proporcional controla el caudal de los pistones del Sistema hidráulico del Pitch, ante una señal de consigna desde el PLC, la electrónica acondiciona y modula la corriente eléctrica para mover proporcionalmente unas válvulas mediante un solenoide. Las válvulas controlan el caudal y la dirección del aceite que entra a los pistones hidráulicos del sistema de control de Pitch.

Figura 3–3 Valvula proporcional ATOS.

La válvula proporcional tiene dos conectores eléctricos, uno de alimentación de 24V para la electrónica de potencia que mueve el solenoide y un conector M12 para la alimentación a 24V del control y la comunicación CANopen.

Vía CANopen, la válvula proporcional recibe desde el PLC:

- Estado de operación (Operativo, preoperativo, stop, etc.)
- Consigna de control, posición de desplazamiento del solenoide.

Y envía información al PLC de los siguientes parámetros:

- Estado de operación(not ready, init, disable, Fault, active, etc)
- Posición actual del solenoide.(retroalimentación).

3.1.2 Sensor de posición MTS Temposonic.

El sensor de posición MTS, son sensores de desplazamiento magnetoestrictivos se basa en el principio de medida por magnetoestricción, que hace que los materiales cambien de forma y orientación en presencia de un campo magnético.
Gracias al principio de funcionamiento de los sensores de desplazamiento magnetoestrictivos, se evita el rozamiento al no haber contacto directo, lo que los hace ideales en aplicaciones donde se realizan millones de ciclos.
Los sensores MTS miden la posición lineal del vástago del cilindro hidráulico, mediante el cual, el PLC calcula la posición angular de las palas.

El sensor MTS esta formado por una varilla detectora, un magneto y la electrónica digital y analógica para el acondicionamiento de señal y comunicación vía CANopen.
El aspecto del sensor MTS se muestra en la siguientes figuras.

Figura 3–4 Sensor MTS temposonic y montaje en piston hidraulico.

El sensor MTS tiene un solo conector M12, para la alimentación a 24V y los canales CAN-H, CAN-L y GND para la comunicación CAN.

Los datos transmitidos por el sensor de posición son los siguientes:

- Valor de la posición.
- Velocidad de desplazamiento.
- Estado de operación.

3.1.3 Sensor de presión TRAFAG.

El sensor es un transductor que genera una señal eléctrica en función de la presión a la que es sometido y digitalizada por un microcontrolador.
Los sensores de presión del buje, miden la presión de gas de los acumuladores hidráulicos. Este sensor a su vez mide también temperatura.
El aspecto del sensor de presión trafag es el siguiente:

Figura 3–5 Sensor de presión Trafag.

El sensor de presión Trafag, tiene un conector M12, para la alimentación a 24V y para los canales CAN-H, CAN-L y GND para la conexión al CAN bus.

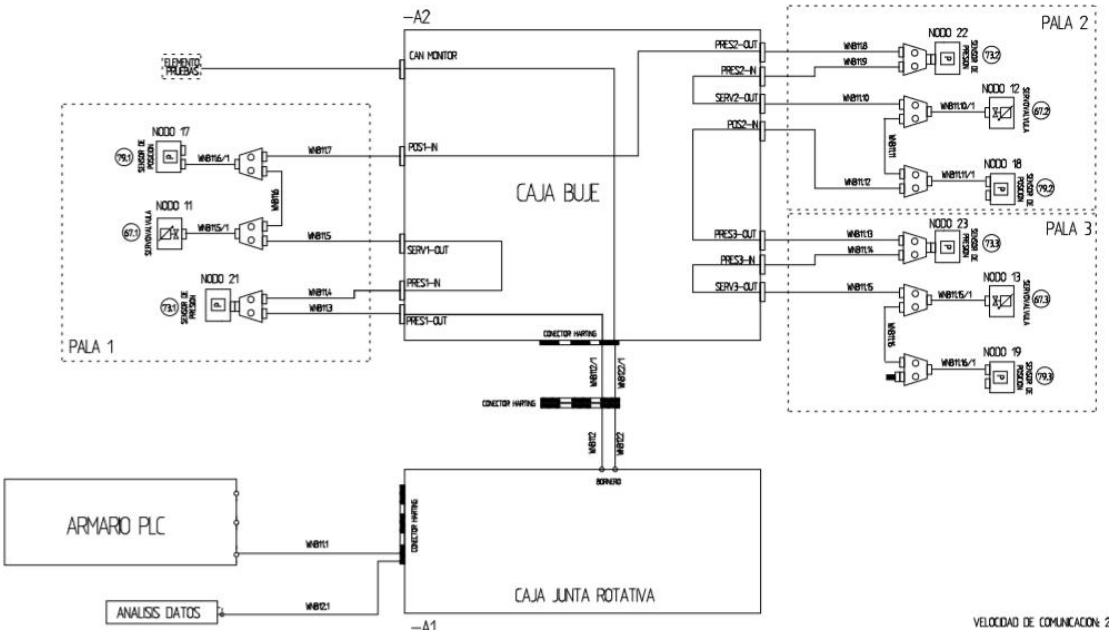
El sensor esta configurado para transmitir por CAN bus:

- Presión (Bares).
- Temperatura (C).

3.1.4 Esquema de conexiones de los dispositivos CANopen.

Los sensores y actuadores CANopen para el sistema hidráulico del pitch, están distribuidos en el buje y conectados a un bus común por el que reciben a su vez alimentación de 24V. La conexión entre la nacelle y el buje es a través de una junta rotativa por el que comunican los canales CAN-H y CAN-L, alimentación 24V, entre otras muchas señales que pertenecen a otros sistemas. El esquema eléctrico de conexiones del sistema de control de pitch se muestra a continuación.

Figura 3–6 Conexiones de los dispositivos CANopen en el buje.



La asignación de nodos en la red CAN del buje es la siguiente:

Tabla 3-1 Asignación de nodos de la red CANbus.

	Dispositivo	Nodo (DEC)
PALA 1	ATOS P1	11
	MTS P1	17
	TRAFAG P1	21
PALA 2	ATOS P2	12
	MTS P2	18
	TRAFAG P2	22
PALA 3	ATOS P3	13
	MTS P3	19
	TRAFAG P3	23

3.2 Configuración de los sensores CANopen.

Ingeniería de Acciona según sus estrategias de control, decide como configurar sus sensores, no es objetivo de este capitulo explicar las razones de la configuración que tienen. Al no tener información de como están configurados los equipos, la configuración de los sensores ha sido obtenida analizando los archivos de configuración del software Cosecan(Software de Glual, fabricante del grupo hidráulico) que es utilizado actualmente para configurar los sensores CANopen. El objetivo es saber como esta configurado el diccionario de objetos de cada sensor y diseñar con el protocolo CANopen, las secuencias necesarias para la configuración de los sensores con el equipo diseñado en este proyecto.

Los dispositivos CANopen del aerogenerador AW3000, ademas del baud rate y el numero de nodo, en sus especificaciones hay una serie de parámetro que deben ser configurados en el diccionario de objetos. Algunos parámetros son propios de cada dispositivo, relacionados con su funcionamiento y otros parámetros están relacionados en como interacciona el dispositivo en la red CANopen. Los parámetros a configurar en un dispositivo CANopen son:

- **Baud rate.** Velocidad de comunicación en kbits/s que debe haber en toda la red CAN.
- **Numero de Nodo**. Numero de identificación único asignado a cada dispositivo.
- **TPDO (Transmision process data objetc).** Es la configuración de los TPDO, que son los datos que transmitirá el sensor, y en donde se configuran el TPDO a utilizar(del 1 al 4), objetos a transmitir, el orden y el modo de transmisión Síncrona/Asíncrona.
- **RPDO(Received process data object).** Es la configuración de los mensajes que podrá recibir el dispositivo y la distribución de los objetos que contiene, en este aerogenerador, solo los ATOS tiene activo el RPDO 2, para recibir desde el PLC, la consigna de control.
- **Configuración de objetos del dispositivo.** En el que se configuran ciertos objetos relacionados con funcionamiento del dispositivo.

El datasheet y los diccionarios de objetos de cada dispositivo se encuentra en en los anexos. Se ha diseñado una tabla que muestra la configuración de cada dispositivo.

3.2.1 Configuración de los Actuadores ATOS.

Tabla 3-2 Configuración CANopen de los actuadores ATOS.

CAN SENSOR CONFIGURATION LIST				
Sensor Name:	ATOS PALA 1,2,3			Descripción
NODO:	11/12/13			Nodos 11,12,13
Bit timing:	250			baud rate 250kbits/s
TPDO 2				
Transmision type:	Sincrona			
Numero Objetos Mapeados	2			
No. de Objeto	Indice	Subindice	No. de bits	Descripción
Objeto 1	x6041	0	16	Estado de operación
Objeto 2	x6301	1	16	Posición de la valvula
Objeto 3				
RPDO 2				
Transmision type:	Sincrona			
Numero Objetos Mapeados	2			
No. de Objeto	Indice	Subindice	No. de bits	Descripción
Objeto 1	x6040	0	16	Estados de operación
Objeto 2	x6300	1	16	Consigna de posición
Objeto 3				
Servicio SDO				
Indice	Subindice	No. de bits	Valor(Hex)	Descripción
x6042	0	8	1	Control desde el bus

Cada válvula proporcional ATOS tiene asignado un numero de nodo, 11,12,13. Los datos transmitidos TPD02 tienen los siguientes COB-ID, 28B,28C,28D. A su vez el Atos tiene configurados el RPDO2 para recibir las consignas de control desde el PLC cuyos mensajes serán transmitidos desde el PLC con las consignas de control, con los COB-ID siguientes, 30B,30C,30D. Ambas tendrán de acuerdo a la configuración la siguiente estructura:

Figura 3–7 Configuración TPDO y RPDO de actuadores ATOS.

COB-ID	RTR	DLC	Byte 0	Byte 1	Byte2	Byte 3
TPDO2 x28B,28C,28D	0	4	DO x6041.0 Estado de operación		DO x6301 Posición de la valvula	

COB-ID	RTR	DLC	Byte 0	Byte 1	Byte2	Byte 3
RPDO2 x30B,30C,30D	0	4	DO x6040.0 Estado de operación		DO x6300 Consigna Posición de la valvula	

3.2.2 Configuración sensor de posición MTS.

Tabla 3-3 Configuración de sensores de posición MTS.

CAN SENSOR CONFIGURATION LIST				
Sensor Name:	MTS PALA 1,2,3			Descripción
NODO:	17/18/19			Nodos 17, 18, 19
Bit timing:	250			Baud rate 250kbits/s
TPDO 1				
Transmision type:	Sincrona/Asincrona			
Numero Objetos Mapeados	3			

No. de Objeto	Indice	Subindice	No. de bits	Descripción
Objeto 1	x6020	1	32	Posición
Objeto 2	x6030	1	16	Velocidad
Objeto 3	x6300	1	8	Estado operación
Objeto 4				

Los sensores MTS nodos 17,18,19 han sido configurados para transmitir datos en el TPDO1, con los COB-ID x191,x192 y x193 los primeros 4 bytes de la trama de datos corresponden a la posición del sensor en mm, los dos bytes siguientes a la velocidad y el ultimo byte es utilizado para transmitir el estado de operación del sensor.
El TPDO1 para los sensores MTS tendría la siguiente estructura:

Figura 3–8 Configuración TPDO sensor MTS.

COB-ID	RTR	DLC	Byte 0	Byte 1	Byte2	Byte 3	Byte4	Byte 5	Byte 6
TPDO1 x191,192,193	0	7	DO x6020.1 - Posición				DO x6030.1- Velocidad		DO x6300.1 Estado

3.2.3 Configuración sensor de presión Trafag.

Tabla 3-4 Configuración sensores Trafag.

CAN SENSOR CONFIGURATION LIST				
Sensor Name:	Carlo Gavazzi			Descripción
NODO:	21/22/23			Nodos 21,22,23
Bit timing:	250			Baud rate 250kbits/s

Servicio SDO				
Indice	Subindice	No. de bits	Valor(Dec)	Descripción
x2100	1	8	17	Mapping PDO1 Mode

En el caso el sensor de presión Trafag, de acuerdo al manual del dispositivo el TPDO no se configura directamente, si no a través de la configuración de un objeto del diccionario con dirección x2100. Donde se configura de acuerdo al manual del dispositivo con los valores de una lista con las distribución del TPDO preprogramadas.

Tabla 3-5 Configuración TPDO1 de los sensores Trafag.

PDO-Mapping-Table

Prepared variable mapping

PDO MappingMode	Entry 1	Entry 2	Entry 3	Entry 4	Size [Bytes]	Event Mask (standard value)
1	P_Int32	P_Status	-	-	5	00010000
16	T_Float	-	-	-	4	00100000
17	P_Int32	T_Int32	-	-	8	00110000
18	P_Int24	T_Int24	-	-	6	00110000

La distribución del TPDO1 para los nodos de los sensores trafag 21,22 y 23 es la siguiente:

Figura 3–9 TPDO Trafag.

COB-ID	RTR	DLC	Byte 0	Byte 1	Byte2	Byte 3	Byte4	Byte 5	Byte 6	Byte 7
TPDO1 x195,196,197	0	8	DO x2110.3 Presión				DO x2111.3 Temperatura			

Todos lo sensores son configurados a un baud rate de 250 kbits/s y la transmisión de TPDOs y RPDOs están configurados como síncrona, lo que significa que los sensores envían sus dato cada vez que el PLC envía un mensaje de sincronía COB-ID x80
Para mejor comprensión de como funciona la comunicación, con la configuración actual de los sensores, la tabla siguiente muestra una fracción de los datos en el bus CAN, capturados con un convertidor USB a CAN IXXAT.

Tabla 3-6 Captura de CAN bus.

COB-ID(HEX)	DLC	DATOS(HEX)	INFORMACIÓN
80	0		Mensaje de sincronia desde el PLC
195	8	CA 3B 03 00 CC 00 00 00	Datos sensor Trafag Nodo 21
191	7	18 3D 04 00 00 00 00	Datos sensor MTS Nodo 17
192	7	21 3C 04 00 00 00 00	Datos sensor MTS Nodo 18
193	7	62 3D 04 00 00 00 00	Datos sensor MTS Nodo 19
196	8	81 3D 03 00 BB 00 00 00	Datos sensor Trafag Nodo 22
197	8	81 3D 03 00 A4 00 00 00	Datos sensor Trafag Nodo 23
28B	4	1F 00 18 FF	Datos Atos Nodo 11
28C	4	1F 00 F0 FE	Datos Atos Nodo 12
28D	4	1F 00 28 FF	Datos Atos Nodo 13
30B	4	00 00 0F FF	Consignas a Atos nodo 11
30C	4	00 00 10 FF	Consignas a Atos nodo 12
30D	4	00 00 0C FF	Consignas a Atos nodo 13

En donde se ve que una vez que el PLC envía el mensaje de sincronía (SYNCRON x80), todos los dispositivos responden con el TPDO configurado. Y el PLC a su vez envía la consignas de control a las tres válvulas proporcionales ATOS.

3.3 Proceso de configuración de los sensores CANopen.

La configuración de los sensores CAN, se hace previa a su instalación y requiere una linea de comunicación única con el dispositivo. El dispositivo si es nuevo, tiene un numero de nodo y baud rate por defecto, o si ha sido utilizado antes, puede haber sido configurado para otro nodo y un baud rate de 250kbits/s usado en la red CAN de Acciona. En cualquiera de los casos para poder iniciar el proceso de configuración es necesario saber el baud rate para poder comunicarse con el dispositivo.

3.3.1 Configuración Nodo y baud rate.

El baud rate y el numero de nodo son las primeras variables a configurar, mediante el protocolo LSS(Layer setting service). La secuencia de configuración del baud rate y el numero de nodo es la siguiente :

Tabla 3-7 Configuración LSS de Baud rate y numero de nodo.

				DATA								
Tx/Rx	COB-ID	RTR	DLC	B0	B1	B2	B3	B4	B5	B6	B7	DESCRIPCION
Tx	0	0	2	x2	0							NMT Command : STOP
Tx	x7E5	0	8	x4	1	0	0	0	0	0	0	LSS Configuration Mode
Tx	x7E5	0	8	x11	Node-ID	0	0	0	0	0	0	LSS Config Node-ID
Rx	x7E4	0	8	x11	0	0	0	0	0	0	0	LSS Node-ID confirmation
Tx	x7E5	0	8	x13	0	Baud Rate	0	0	0	0	0	LSS Set Baud rate
Rx	x7E4	0	8	x13	0	0	0	0	0	0	0	LSS Baud rate confirmation
Tx	x7E5	0	8	x17	0	0	0	0	0	0	0	LSS Save parameters
Rx	x7E4	0	8	x17	0	0	0	0	0	0	0	LSS Save confirmation
Tx	x7E5	0	8	x4	0	0	0	0	0	0	0	LSS Operation mode

Donde Tx es el comando transmitido desde el Maestro y Rx el mensaje recibido desde el esclavo. El protocolo LSS solo esta activo en estado STOP, por lo que con el protocolo NMT se cambia el estado del sensor a Stop. Y mediante LSS se pasa al modo de configuración. Donde las variables Node-ID y baud rate son remplazadas con los valores de la configuración.

Configurado el Nodo y el baud rate, es necesario reiniciar el dispositivo, para que los cambios se hagan efectivos en el equipo. Para reiniciar es posible hacerlo apagando y encendiendo el equipo o mediante un comando NMT.

3.3.2 Configuración TPDOs y RPDOs.

La siguiente configuración es la de los PDOs, que son los datos transmitidos y recibidos por el dispositivo. Para configurarlo se utiliza el protocolo SDO(service data object) utilizado para leer y escribir sobre el diccionario de objetos específicamente sobre el área de perfil de comunicación(Communication profile area) en los registros relativos a cualquiera de los cuatro TPDOs y RPDOs.

Para utilizar el protocolo SDO debe conocerse el numero de nodo que el dispositivo tiene configurado, en este caso el nodo asignado mediante LSS(Layer setting service).

Los registros relativos a los TPDOs y RPDOs en el diccionario de objetos que deben ser configurados, tiene el siguiente formato.

Tabla 3-8 Registros TPDOs y RPDOs.

Transmit PDO Communication settings					
Index	Subindex	Parameter Name	Comment	Acces type	Data Format
x1800	x1	TPDO1 COB-ID	COB-ID x180+Node-ID	RW	U32
	x2	TPDO1 Transmision Type	Synchronus x00/Asynchronus xFF	RW	U8

Transmit PDO Mapping					
Index	Subindex	Parameter Name	Comment	Acces type	Data Format
x1A00	x0	TPDO num of mapped object	Number of Mapped objects	RW	U8
	x1	Mapping Object 1	1st Mapping parameter	RW	U?
	x2	Mapping object 2	2nd Mapping parameter	RW	U?
	x3	Mapping object 3	3rd Mapping parameter	RW	U?
	x4	Mapping object 4	4th Mapping parameter	RW	U?

El mostrado, es para el TPDO1. En los otros TPDOs y RPDOs los registros son iguales salvo los indices que tienen diferente asignación.

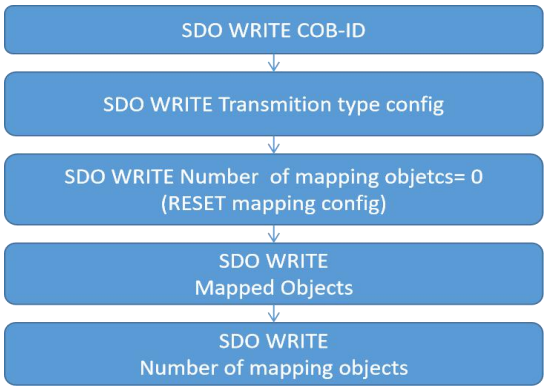
En el “Transmit PDO communication settings” son configurados el COB-ID del TPDO-N y el tipo de transmisión del TPDO, sincrona (x00) o asincrona (xFF).

Y en el “transmit PDO mapping” se configuran los objetos del diccionario que serán transmitidos en el TPDO, en el orden asignado por el numero de objeto de la lista y con la asignación del espacio ocupado por cada objeto.

El nodo para aceptar los comandos del protocolo SDO debe estar en el estado, operativo o preoperativo. Estado en el que estará después del reset posterior a la configuración del nodo y el baud rate.

La secuencia para programar los TPDOS mediante el protocolo SDO es la siguiente:

Figura 3–10 Secuencia de programación de PDOs.



En términos de código SDO para el Nodo N seria el siguiente, considerando para mejor comprensión el TPDO1. Los comandos precedidos por Tx, son los comandos que envía el equipo de configuración y los Rx las respuestas del dispositivo que esta siendo configurado.

Tabla 3-9 Secuencia de programación de TPDOs.

				DATA								
Tx/Rx	COB-ID	RTR	DLC	B0	B1	B2	B3	B4	B5	B6	B7	DESCRIPTION
Tx	x600+Node	0	8	22	00	18	1	PDO + Node				COB-ID Configuration
Rx	x580+Node	0	8	60	00	18	1	0	0	0	0	Write Confirmation
Tx	x600+Node	0	8	22	00	1A	2	0	0	0	0	PDO transmision type SYNC(x00)
Rx	x580+Node	0	8	60	00	1A	2	0	0	0	0	Write Confirmation
Tx	x600+Node	0	8	22	00	1A	0	0	0	0	0	Num. Mapping Objects = 0(Reset)
Rx	x580+Node	0	8	60	00	1A	0	0	0	0	0	Write Confirmation
Tx	x600+Node	0	8	22	00	1A	1	Object Index				Mapping Object 1
Rx	x580+Node	0	8	60	00	1A	1	0	0	0	0	Write Confirmation
...												...
Tx	x600+Node	0	8	22	00	1A	n	Object Index				Mapping Object N
Rx	x580+Node	0	8	60	00	1A	n	0	0	0	0	Write Confirmation
Tx	x600+Node	0	8	22	00	1A	0	N	0	0	0	Num. Mapping Objects = N
Rx	x580+Node	0	8	60	00	1A	0	0	0	0	0	Write Confirmation

SDO Command	x22 Write over DO
	x60 Write confirmation

Para cualquiera otro PDO, se ajustan los indices que correspondan al tipo y numero de PDO(RPDO o TPDO)

3.3.3 Configuración de objetos del diccionario.

Para configurar otros objetos del diccionario, relacionados con el funcionamiento del dispositivo, se utiliza el protocolo SDO utilizado anteriormente para la configuración de los PDOs. La secuencia siguiente se repite por cada objeto modificado.

Tabla 3-10 Configuración SDOs.

				DATA(HEX)								
Tx/Rx	COB-ID	RTR	DLC	B0	B1	B2	B3	B4	B5	B6	B7	DESCRIPTION
Tx	600+Node	0	8	22	Index		Subindex	DATA				Write over DO
Rx	580+Node	0	8	60	Index		Subindex	0	0	0	0	Write Confirmation

SDO Comm	x22 Write over DO
	x60 Write confirmation

3.3.4 Salvar Configuración.

Las modificaciones mediante SDO, están activas mientras el dispositivo este encendido, si el dispositivo tiene un reinicio por apagado o por comando NMT, los cambios efectuados se perderán. Para hacer efectivos los cambios en el dispositivo de forma permanente, es necesario escribir sobre el objeto x1010.1, los caracteres en código ASCII 'save'(6C,6F,61,65), para conservar los cambios aplicados en la eprom del dispositivo.

Por lo que el código final del proceso de configuración es el siguiente.

Tabla 3-11 SDO Salvar configuración.

				DATA(HEX)								
Tx/Rx	COB-ID	RTR	DLC	B0	B1	B2	B3	B4	B5	B6	B7	DESCRIPTION
Tx	x600+Node	0	8	22	10	10	1	73	61	76	65	WRITE 'save' (x73,x61,x76,x65)
Rx	x580+Node	0	8	60	10	10	1	0	0	0	0	Write Confirmation

SDO Command	x22 Write over DO
	x60 Write confirmation

Finalizada la configuración se efectúa un reset y es recomendable comprobar que la configuración es correcta.

3.3.5 Restaurar valores por defecto.

El tipo de configuración que se realiza mediante este proceso, solo modifica los objetos que estén en la lista de configuración y se asume que el resto de los objetos del diccionario conservan su valor por defecto. Si alguno de esos valores ha sido modificado mediante algún otro software, el código de configuración no modificara aquellos objetos que no estén en su lista. Una característica aprovechable para la configuración de los dispositivo es la posibilidad de recuperar los valores por defecto del dispositivo. Por lo que en este proyecto se utiliza esta función al inicio de la configuración. Por lo que al inicio de la configuración se escribe sobre el objeto 1011.1 la palabra 'load' en código ASCII (x6C,x6F,x61,x64).

Tabla 3-12 SDO Restaurar valores por defecto.

				DATA(HEX)								
Tx/Rx	COB-ID	RTR	DLC	B0	B1	B2	B3	B4	B5	B6	B7	DESCRIPTION
Tx	x600+Node	0	8	22	11	10	1	6C	6F	61	64	WRITE 'load' (x6C,x6F,x61,x64)
Rx	x580+Node	0	8	60	11	10	1	0	0	0	0	Write Confirmation

SDO Command	x22 Write over DO
	x60 Write confirmation

3.4 Especificaciones del equipo de configuración de sensores CANopen.

Definido el procedimiento para la configuración de los dispositivos CAN del aerogenerador AW3000 con el protocolo CANopen, se describen las especificaciones necesarias que se buscan en el equipo de configuración CANopen.

El equipo diseñado busca tener las siguientes características :

Construcción:

- **Portabilidad**. Que pueda sostenerse con una mano.
- **Resistencia.** Con una carcasa resistente a posibles golpes que puedan darse durante las tareas comunes de mantenimiento.

Periféricos:

- **Puerto USB.** para programación del microcontrolador y posibilidad de conexión con un ordenador para futuras aplicaciones.
- **Conector DC.** Para alimentación de 24V desde fuente externa.
- **Conector M12, 5 polos, hembra.** Para comunicación conexión con el bus CAN en el interior del buje.
- **Conector DB9 macho y hembra.** Para conexión con el cable CAN del PLC de la nacelle.
- **Memoria SD.** Para guardar los archivos de configuración de los dispositivos CANopen, y para otros posibles usos futuros.
- **Pantalla LCD gráfica.** Para visualización de las diferentes funciones del equipo.
- **Mando.** 3 push button para la navegación entre menús del dispositivo. (Arriba,abajo, Enter)

Hardware:

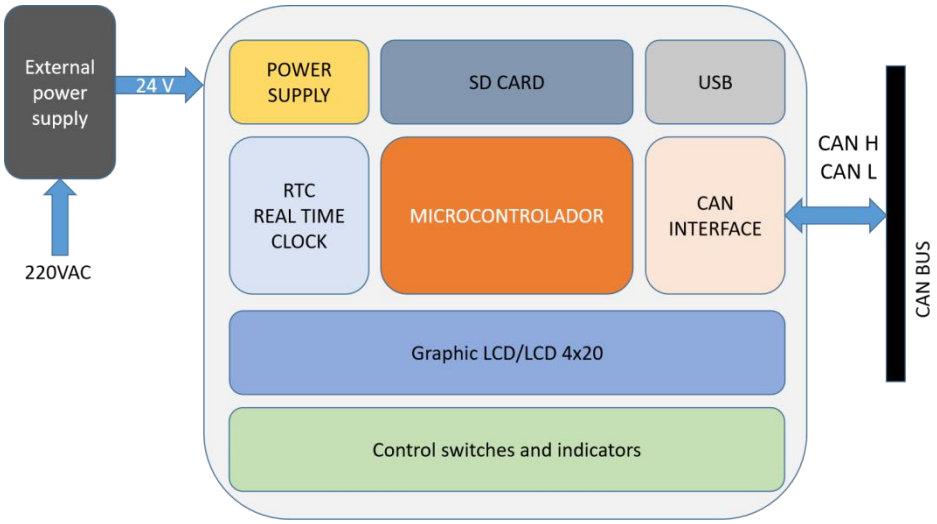
Microcontrolador. Basado en arduino por ser un modulo de desarrollo de bajo coste y con librerías open source para el control de diversos dispositivos.
Controlador CAN. Para la interface lógica de control con el bus CAN.
Reloj en tiempo real(RTC). Para aplicaciones futuras de registro de eventos.

3.5 Diseño del Hardware.

Para el diseño del equipo de configuración CAN, se utilizan tres programas de diseño, solidworks para el diseño mecánico, Multisim y Ultiboard de National Instruments para el esquemático y el diseño del circuito impreso.

De las especificaciones definidas para el equipo, el Hardware puede dividirse en bloques funcionales, mostrados en la Figura siguiente.

Figura 3–11 Bloques funcionales del equipo.



Del diagrama de bloques del equipo, la fuente externa, es una fuente comercial de 220V a 24V para alimentar el equipo, donde una Fuente interna acondiciona los 24V para los voltajes requeridos en la electrónica interna 5V y 3.3V. El microcontrolador gestiona los periféricos, los de nivel de usuario Interfaz gráfica y mando, interfaz de almacenamiento en la SD, los de comunicación con el BUS CAN, de reloj en tiempo real, y puerto USB. El diseño de cada bloque se explican a mayor detalle en los siguientes párrafos.

3.5.1 Microcontrolador.

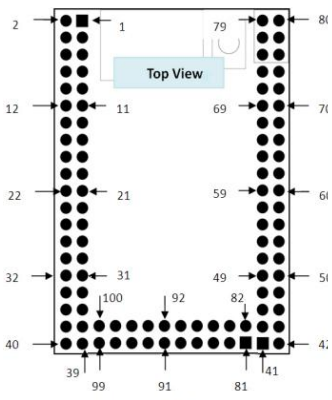
De las múltiples plataformas de desarrollo de microcontrolador, se opta por Arduino por ser una plataforma de desarrollo open source muy conocida, económica, con una amplia gama de microcontroladores soportados y bibliotecas de librerías para diferentes interfaces, lo que ahorra tiempo en el desarrollo de cualquier equipo electrónico.

El microcontrolador seleccionado es el ATMEGA2560 del fabricante ATMEL, que por sus características, cubre el numero de puertos digitales y puertos de comunicación serie, requeridos para el control del equipo. Para facilitar el montaje y fácil remplazo del microcontrolador en caso de avería, se utiliza el modulo MEGA2560 CORE, del fabricante Inahos. Es básicamente una versión mas compacta del Arduino Mega. Que permite por su tamaño, integrarse en nuevos diseños, que facilita la integración del microcontrolador con conectores en vez de integrados de montaje superficial.

Las características del microcontrolador y el aspecto del mismo se muestran a continuación.

Figura 3–12 Especificaciones MEGA2560.

Processor	ATMEGA2560
Operative Voltaje	3/5V
CPU SPEED	16 MHz
ANALOG IN	16
Digital IO/PWM	54/15
EEPROM	4 kB
SRAM	8 kB
Flash	256 kB
UART	4
Serial communication	SPI, I2C



Pin	Function /MCU	Arduino Mega2560	Pin	Function /MCU	Arduino Mega2560	Pin	Function /MCU	Arduino Mega2560
1	VCC		40	GND		79	PB3	D50
2	VCC		41	AGND		80	VCC	
3	PG5	D4	42	AGND		81	VCC	
4	PE0	RXD	43	ADC0	A0	82	VCC	
5	PE1	TXD	44	VREF		83	PC7	D30
6	PE3	PWM/D5	45	ADC2	A2	84	PC6	D31
7	PE4	PWM/D2	46	ADC1	A1	85	PC5	D32
8	PE5	PWM/D3	47	ADC4	A4	86	PC4	D33
9	PE6		48	ADC3	A3	87	PC3	D34
10	PE7		49	ADC6	A6	88	PC2	D35
11	TXD3	D14	50	ADC5	A5	89	PC1	D36
12	RXD3	D15	51	ADC8	A8	90	PC0	D37
13	TXD2	D16	52	ADC7	A7	91	PG1	D40
14	RXD2	D17	53	ADC10	A10	92	PG0	D41
15	TXD1	D18	54	ADC9	A9	93	PD7	D38
16	RXD1	D19	55	ADC12	A12	94	PD6	
17	SDA	D20	56	ADC11	A11	95	PD5	
18	SCL	D21	57	ADC14	A14	96	PD4	
19	PH2		58	ADC13	A13	97	PG4	
20	PH3	PWM/D6	59	PJ2		98	reserv	
21	PH4	PWM/D7	60	ADC15	A15	99	GND	
22	PH5	PWM/D8	61	PJ4		100	GND	
23	PH6	PWM/D9	62	PJ3				
24	PB0	53	63	PJ6				
25	PB4	PWM/D10	64	PJ5				
26	PB5	PWM/D11	65	PA7	D29			
27	PB6	PWM/D12	66	PG2	D39			
28	PB7	PWM/D13	67	PA5	D27			
29	PL0	D49	68	PA6	D28			
30	PL1	D48	69	PA3	D25			
31	PL2	D47	70	PA4	D26			
32	PL3	PWM/D46	71	PA1	D23			
33	PL4	PWM/D45	72	PA2	D24			
34	PL5	PWM/D44	73	PJ7				
35	PL6	43	74	PA0	D22			
36	PL7	42	75	RESET				
37	PH7		76	GND				
38	PG3		77	PB1	D52			
39	GND		78	PB2	D51			

3.5.2 Controlador e interface CAN.

El siguiente elemento en orden de importancia es el controlador CAN, núcleo del proyecto. De los diferentes controladores CAN del mercado se opto por el controlador MCP2515. Controlador dedicado de CAN bus del fabricante Microchip que tiene las siguientes especificaciones:

- CAN v2.0B hasta 1Mb/s
- SPI de alta velocidad hasta (10 MHz)
- Voltaje de alimentación 2.7-5.5V
- Consumo de corriente 5mA(típico)

El MCP2515 se conecta al BUS can a través de un transductor CAN, que transforma los niveles TTL de los canales TX/RX utilizado por el controlador, a los niveles de tensión diferencial usados por CAN BUS. El transductor utilizado es el MCP2551 del fabricante Microchip. El pinout para ambos circuitos integrados se muestran a continuación.

Figura 3–13 Pinout del controlador y transductor CAN.

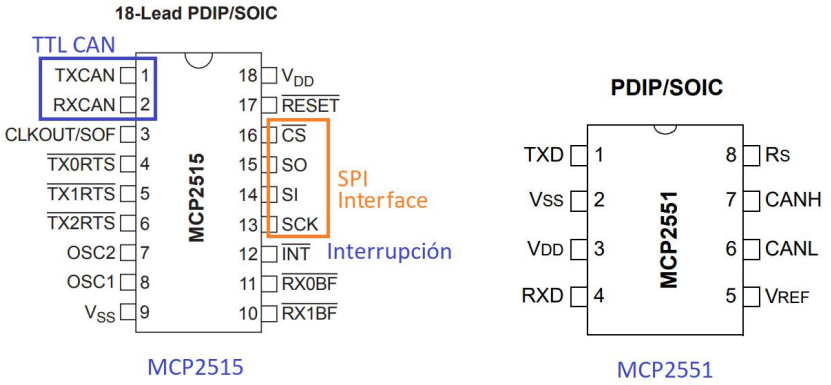
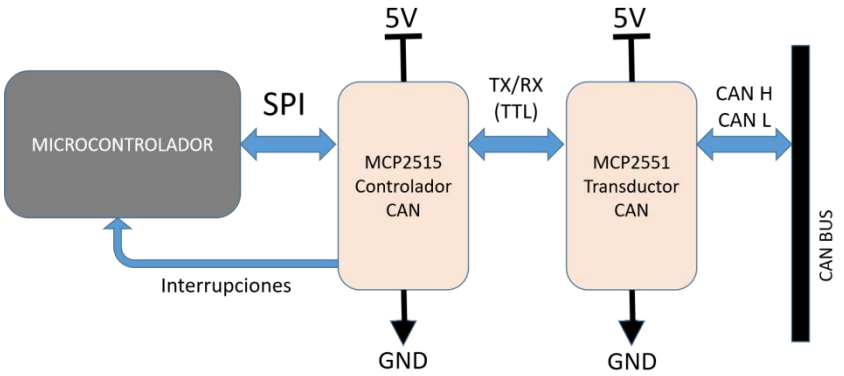


Figura 3–14 Diagrama de bloques del circuito de CAN bus.



El controlador MCP2515, es controlado por el microcontrolador por SPI. Sus salidas de interrupción son utilizadas por el microcontrolador para detectar recepción de mensajes.

3.5.3 Interfaz gráfica.

Para la interfaz gráfica se opto por un modulo HMI de 3.5” con pantalla TFT y panel táctil(no utilizado en el proyecto), modelo Nextion NX4832T035 del fabricante Itead.
Es una pantalla de bajo costo, con hardware con microcontrolador dedicado, cuya interfaz gráfica es diseñada en un ordenador mediante un software, para descargarlo en la memoria flash de la pantalla. Cuenta con un puerto UART para comunicación con el microcontrolador. La ventaja de utilizar este tipo de pantalla es que al tener hardware dedicado todo el procesamiento gráfico es procesado por la pantalla, liberando al microcontrolador de las funciones gráficas, limitándose a gestionar la pantalla mediante el envío de ordenes a través del puerto UART.

Figura 3-15 HMI 3.5" Nextion NX4832TO35.

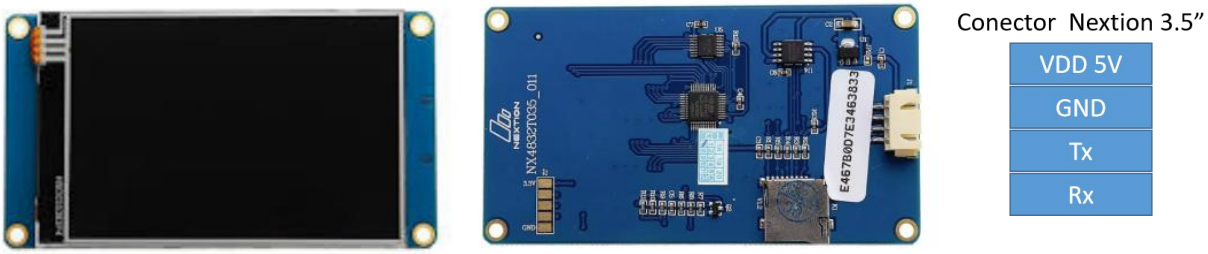
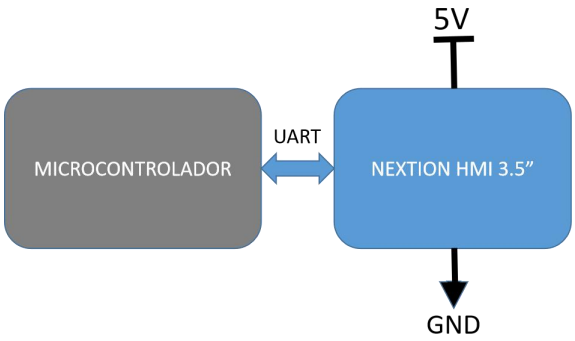


Figura 3-16 Diagrama de conexiones de la interfaz gráfica.



3.5.4 Mando.

El mando sera utilizado para navegar a través de los menús mostrados en la interfaz gráfica. Se utilizara push buttons para las funciones UP, DOWN y ENTER.
Cada push button, esta conectado a una entrada digital del microcontrolador y se utiliza una sola interrupción del micro para indicar que un push button ha sido presionado.

Figura 3-17 Push Buttons y circuito para

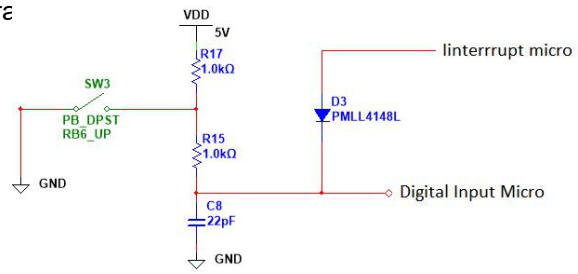
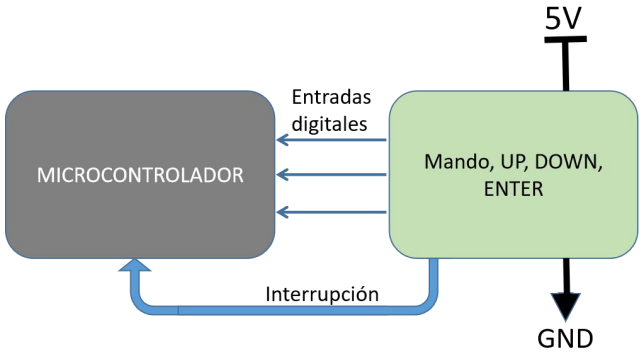


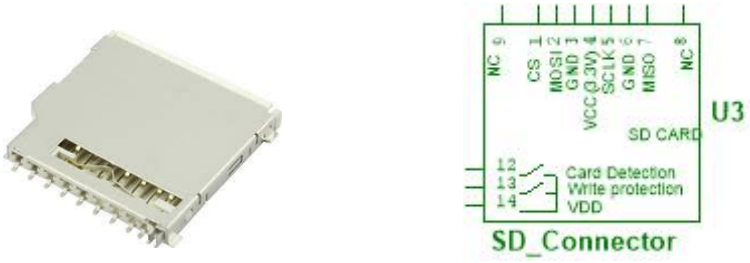
Figura 3-18 Diagrama de conexiones del mando con el microcontrolador.



3.5.5 Lector de memoria SD(SD CARD).

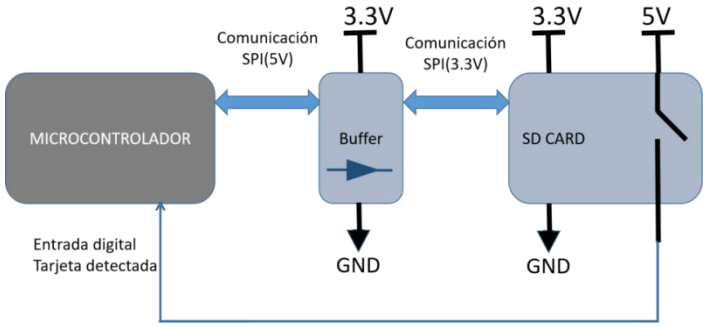
El equipo incluye un lector de memoria SD, donde estarán guardados los archivos de configuración CAN de los sensores. La comunicación con el microcontrolador de la memoria SD es a través del puerto SPI. Se utiliza ademas una entrada digital en el microcontrolador para detectar que la tarjeta esta insertada en el lector. La memoria SD se alimenta con 3.3V.

Figura 3-19 Lector de memoria SD y señales.



El nivel de voltaje de las señales lógicas de la memoria SD es de 3.3V máximo, las del microcontrolador en cambio son de 5V. Conectarlas significa dañar la tarjeta SD, por lo que es necesario un circuito intermedio que acondicione a 3.3V las señales entradas digitales de la SD. El circuito integrado para hacerlo es un buffer 74HC4050, alimentado a3.3V.

Figura 3-20 Diagrama a bloques del lector SD con el microcontrolador.

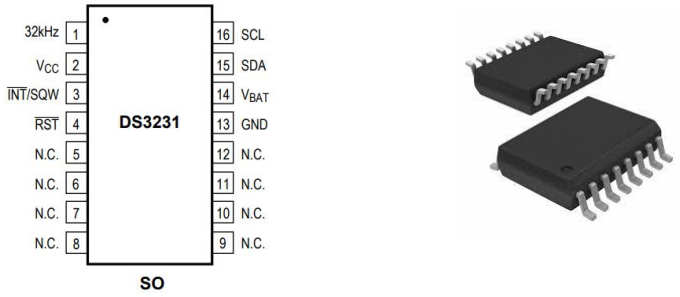


3.5.6 Circuito de reloj en tiempo real, RTC(Real Time Clock).

Para el circuito del reloj en tiempo real se utiliza el integrado DS3231 del fabricante Maxim. Hay varios integrados de otros fabricantes que cumplen con esta tarea, la razón de su elección, es que es un circuito integrado con librerías en el IDE de Arduino, lo que reduce el tiempo de implementación en la programación del microcontrolador.

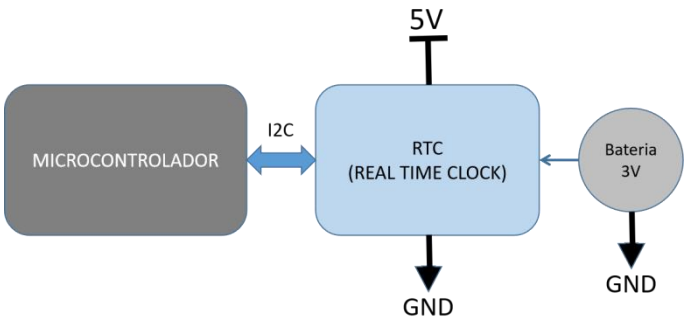
El DS3231 es un reloj en tiempo real preciso con comunicación I2C, que mantiene actualizado el registro de Día, Mes, año, horas, minutos y segundos. Corrigiendo automáticamente la fecha en años bisiestos. Soporta formato de hora de 24Hr o 12Hr con indicador AM/PM. Cuenta con 2 alarmas programables y se alimenta con voltajes entre 2.3-5.5V.

Figura 3-21 Pin out DS3231.



Para la actualización de la hora correcta el DS3230 debe estar siempre alimentado, por ello cuenta con un pin de entrada para conexión a una batería. Cuando pierda la alimentación de 5V durante el apagado del equipo, la batería mantiene las funciones mínimas del integrado para mantener Fecha y Hora actualizadas.
El reloj en tiempo real no es utilizado en esta etapa del proyecto, pero ha sido instalado para funciones adicionales futuras.

Figura 3-22 Diagrama a bloques circuito de reloj en tiempo real.



3.5.7 Puerto USB.

El modulo MEGA2560 CORE no cuenta con Puerto USB, como el Arduino MEGA. Para dotar al equipo de un puerto USB tipo B, se utiliza uno de los puertos UART

conectado al integrado FT232 del fabricante FTDI. Que es un convertidor UART a USB. A través de este puerto se programara el microcontolador y en aplicaciones futuras permitirá la conexión a un ordenador.

Figura 3-23 Pin out FT232 y puerto UART del MEGA2560 CORE.

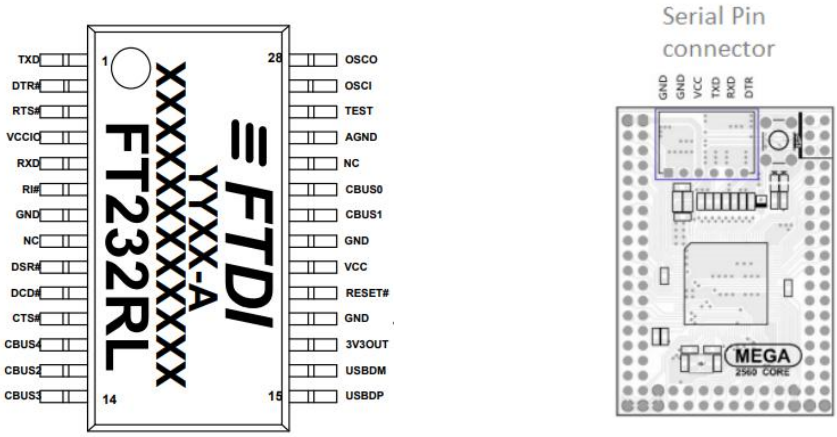
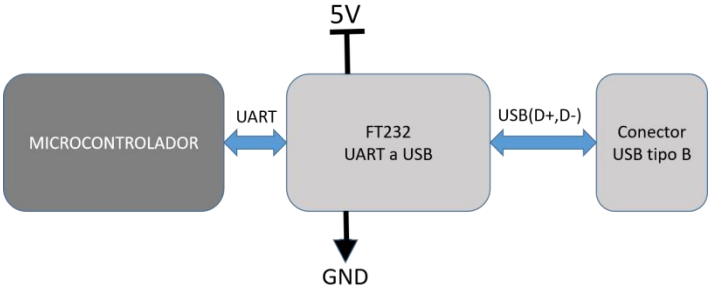


Figura 3-24 Conexión puerto USB con el microcontrolador.

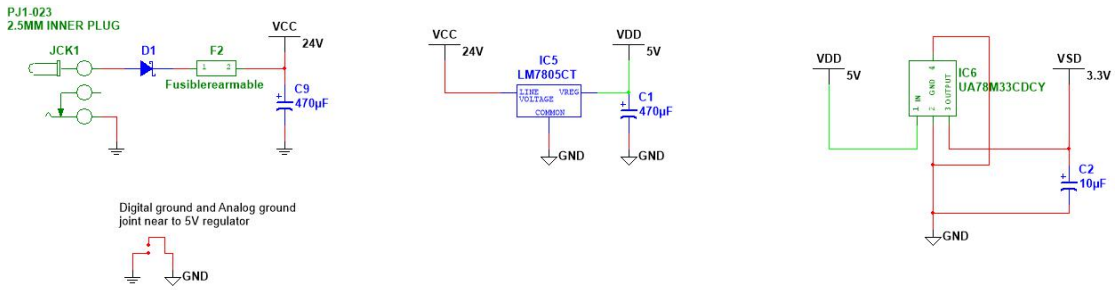


3.5.8 Fuente de alimentación interna y externa (power supply).

Las tensiones requeridas de los diferentes circuitos del equipo son 24V(3A) para alimentación de los sensores CAN a travesó del conector CAN M12 5 polos, 5V (1A) para la electrónica control y 3.3V para el lector de memoria SD.
Para los 24 V se utiliza un adaptador de corriente comercial de 220VAC a 24V. Para los 5V, un regulador conmutado de tracopower TSR-2450, un regulador compatible con la distribución de las terminales de los reguladores lineales de la familia LM78XX. Con corriente máxima de 1A. La ventaja el TSR-2450 es que son reguladores conmutados, mas eficientes, no requieren disipador y son compactos. Por otro lado son dispositivos bastante mas caros (LM7805: 0.5€ vs TSR-2450: 5.5€. RSOnline.com).

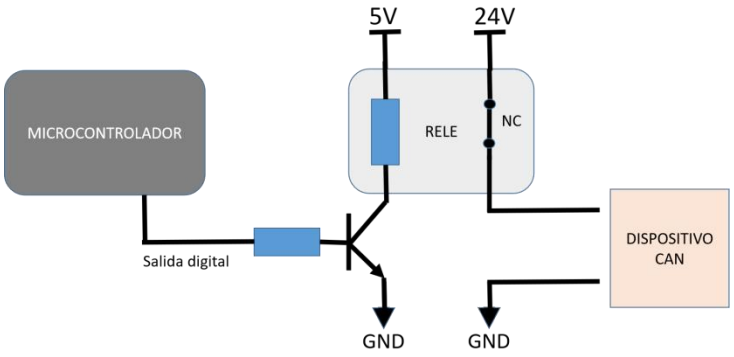
Para el regulador de tensión de 3.3V se utiliza un regulador lineal el UA78M33CDCY con corriente máxima de salida de 500mA suficiente par la alimentación del lector de memorias SD.

Figura 3-25 Esquema electronico de la fuente de alimentación.



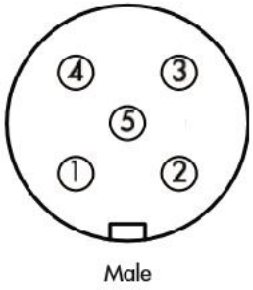
Los 24V para alimentar los Dispositivos CAN por el conector M12 de 5 polos van a través de un relé activado por el microcontrolador, de esta forma se controla el encendido y apagado del dispositivo CAN. Este relé es utilizado durante el proceso de configuración del dispositivo CAN para apagarlo temporalmente y provocar durante el encendido el reinicio del dispositivo con la nueva configuración.

Figura 3-26 Desconexión de la alimentación del dispositivo CANopen.



El equipo esta pensado también para ser alimentado a través del conector CAN M12 de 5 polos, pudiéndose prescindir del adaptador externo de 24V. En pruebas dentro de la red CAN bus del rotor. Conexión para futuras nuevas aplicaciones.

Figura 3.1 distribución de terminales del conector M12, Pin2: 24V Pin 3:GND.

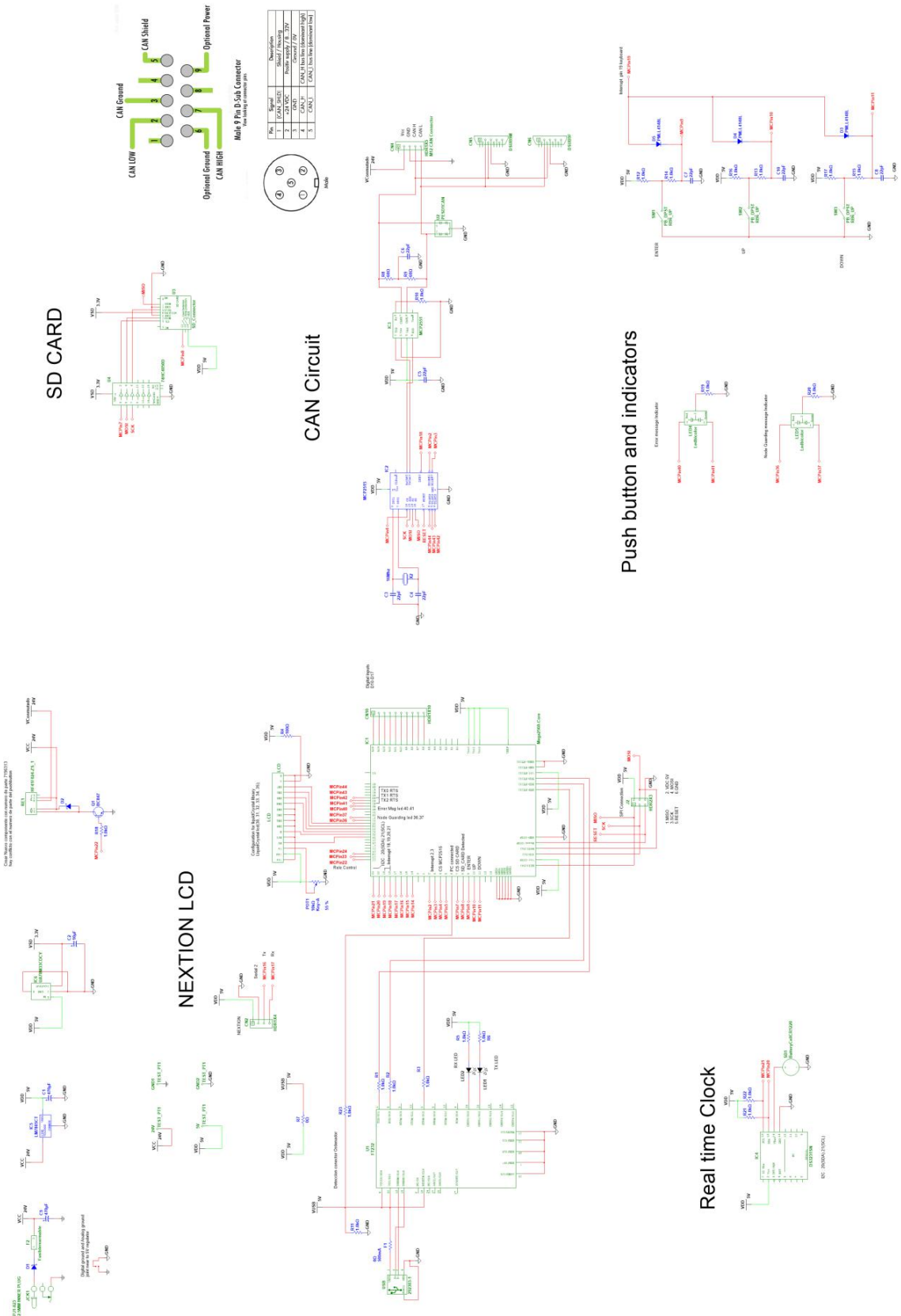


Pin	Signal	Description
1	(CAN_SHLD)	Shield / Housing
2	+24 VDC	Positiv supply / 8...32V
3	GND	Ground / 0V
4	CAN_H	CAN_H bus line (dominant high)
5	CAN_L	CAN_L bus line (dominant low)

3.5.9 Esquema electrónico del equipo de configuración CANopen.

Definidos los bloques funcionales, el esquema electrónico del equipo se muestra en la siguiente pagina.

Figura 3-27 Esquema electrónico del equipo de configuración CANopen.



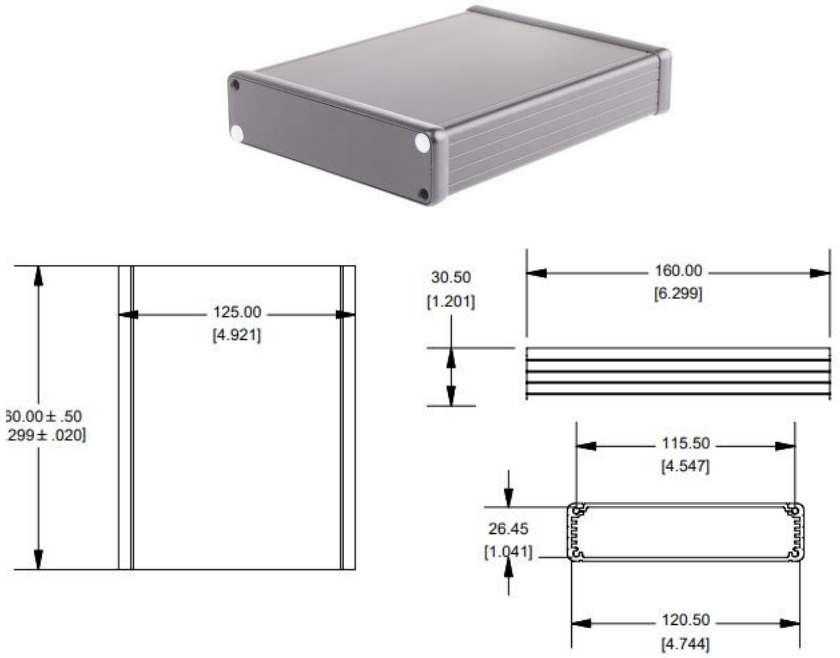
3.6 Diseño Mecánico.

Previo al diseño del circuito impreso (PCB), deben definirse las dimensiones del PCB, restricciones mecánicas, distribución de los elementos inamovibles de la interfaz (mando, pantalla y puertos.).

Las dimensiones del circuito impreso, dependen de la caja que se vaya a utilizar. De las cajas comerciales se busco una que ofreciera resistencia mecánica, y que a su vez tuviera un diseño agradable, que denote calidad de construcción. El ancho de la caja esta condicionado por el ancho de la pantalla y el largo, por la altura de la pantalla mas un espacio suficiente para el mando.

La caja seleccionada es de aluminio de la marca Hammond P/N:1455P1601BK de color negro, una caja robusta para el ámbito de trabajo.

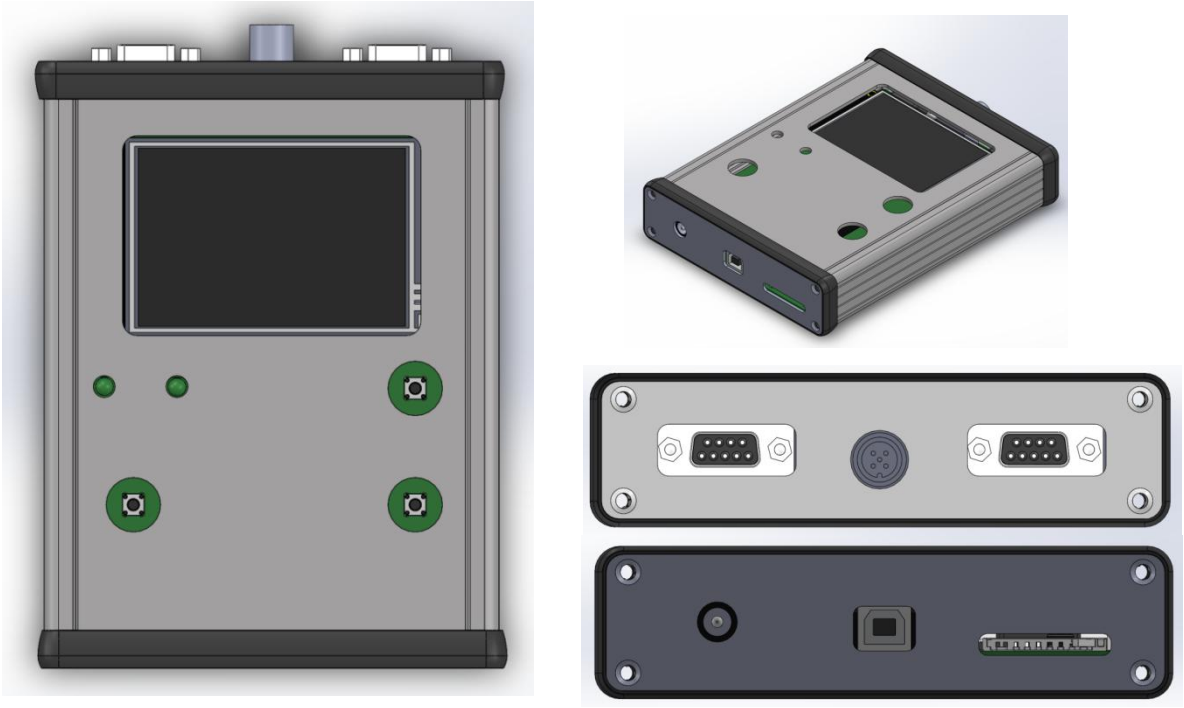
Figura 3-28 Caja de aluminio Hammond P/N:1455P160BK.



La caja tiene ranuras laterales internas para el montaje de un PCB de 1.6mm de espesor. En base a los archivos 3D de fabricante y modelos 3D de los componentes principales(Algunos descargados desde la pagina del fabricante y otros creados al no existir modelo 3D). Mediante Solid Works se plantea la distribución de las interfaces, Pantalla, mando (UP,DOWN, ENTER), conectores CAN (M12 5 polos hembra, DB9 macho, DB9 hembra), USB tipo B, lector de memorias SD y conector DC hembra para la entrada de 24V y dos leds.

Las siguientes figuras muestran el modelo 3D del equipo y la distribución propuesta de los elementos principales.

Figura 3-29 Diseño mecánico del equipo de configuración CANopen.



En el frontal, la pantalla ocupa la parte superior y los tres “push button” del mando se distribuyen en la mitad inferior de acuerdo a su función “UP y DOWN” a la derecha y “Enter” a la izquierda, junto con los dos indicadores led. En el lateral superior se encuentran localizados las conexiones al BUS CAN con los conectores DB9 Macho/Hembra y el conector M12 hembra de 5 polos. En el lateral inferior de derecha a izquierda, el lector de memoria SD, el puerto USB tipo B y conector DC. Con la distribución propuesta la posición del modulo MEGA2560 CORE es la mostrada en los dibujos siguientes, debajo de la pantalla y en el centro.

Figura 3-30 Posición del MEGA2560 CORE.

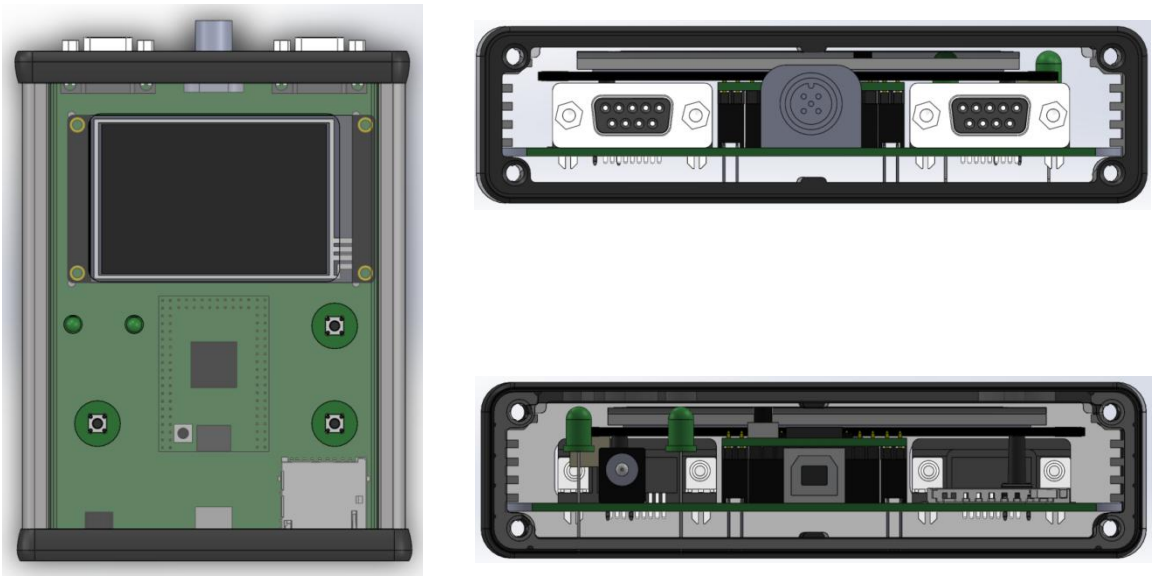
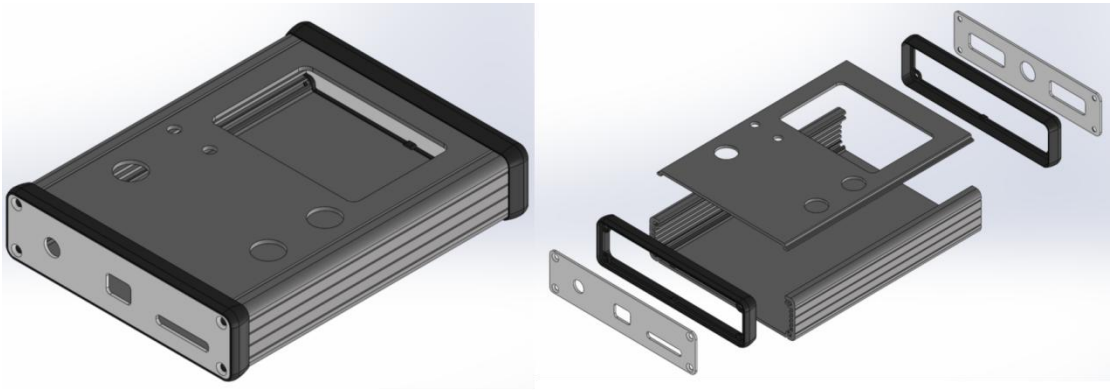


Figura 3-31 Mecanizado y explosionado de la caja.

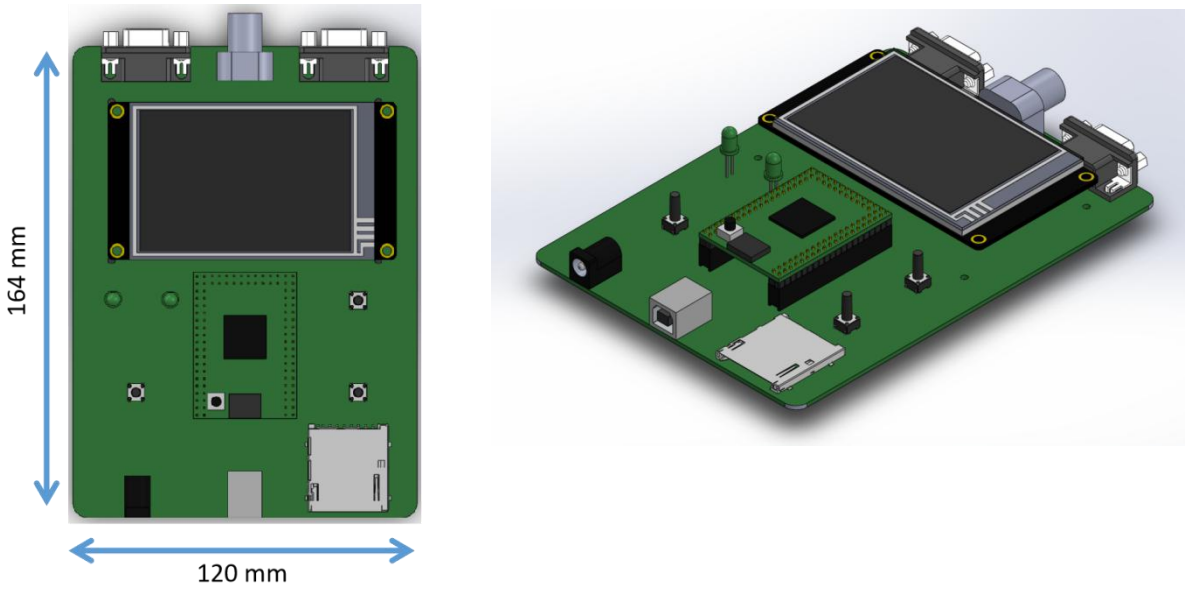


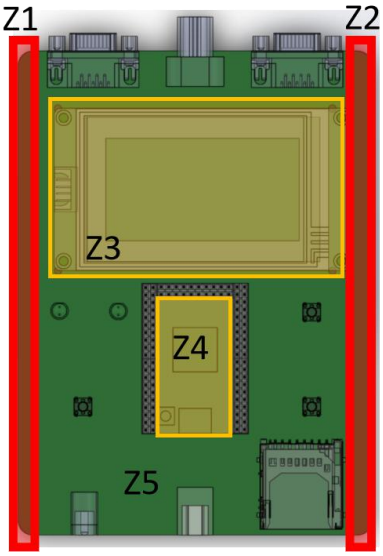
En los anexos se encuentran los planos del las tres piezas de la caja a mecanizar.

3.7 Diseño del circuito impreso (PCB).

Definida la distribución de puertos e interfaces del equipo, las posiciones quedan fijas y el resto de los componentes del circuito electrónico ocupan los espacios libres, cumpliendo con las restricciones de montaje, las reglas de DRC(desing rule Check) que definen distancia entre pistas, entre componentes, ancho de pistas etc.

Figura 3-32 Distribución del PCB y restricciones de montaje.



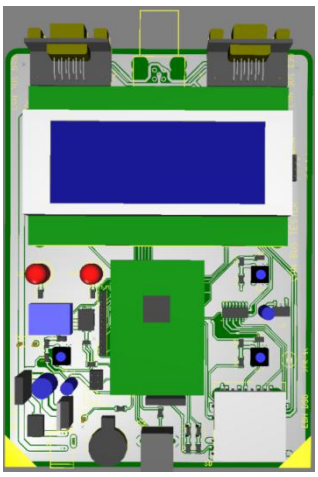


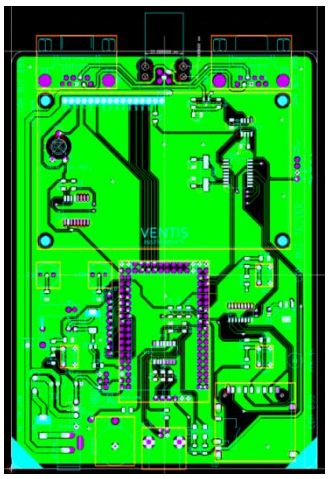
Z1 y Z2	4mm del borde, Zona para insercion del PCB en los carriles internos laterales de la caja, zona prohibida para pistas y montaje de componentes.
Z3	Zona debajo de la pantalla, con restricción de montaje en altura máxima 6mm
Z4	Zona debajo del MEGA2560 CORE, restriccion de montaje en altura máxima 7mm
Z5	Resto de la placa con restricción de montaje en altura de 18 mm máximo, salvo para los push button.

El circuito impreso ha sido diseñado con Ultibord, en base al esquema electrónico generado en Multisim ambos software de National Instruments.
Se han tomado ciertas consideraciones en el diseño:

- Dos planos de tierra separados, uno para los circuitos de 24V y otro para los circuitos digitales (5V y 3.3V). Los planos solo están unidos en un solo punto cercano al regulador conmutado de 5V. Esto con la finalidad de proteger de descargas electrostáticas los circuitos digitales.
- Circuito impreso de doble cara.
- Ancho mínimo de pistas 0.25mm.
- Componentes para montaje SMD y Through hole solo por el part side.
- Placa diseñada para proceso de soldadura DIP para componentes Through hole y reflow para los componentes SMD. (El prototipo es montado y soldado a mano).

Figura 3-33 Diseño de circuito impreso.





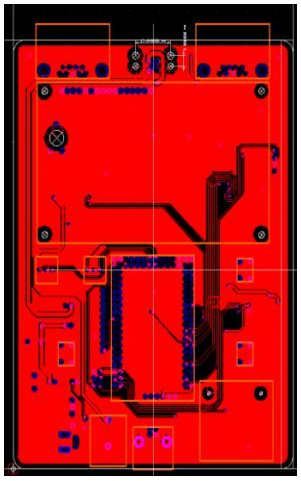
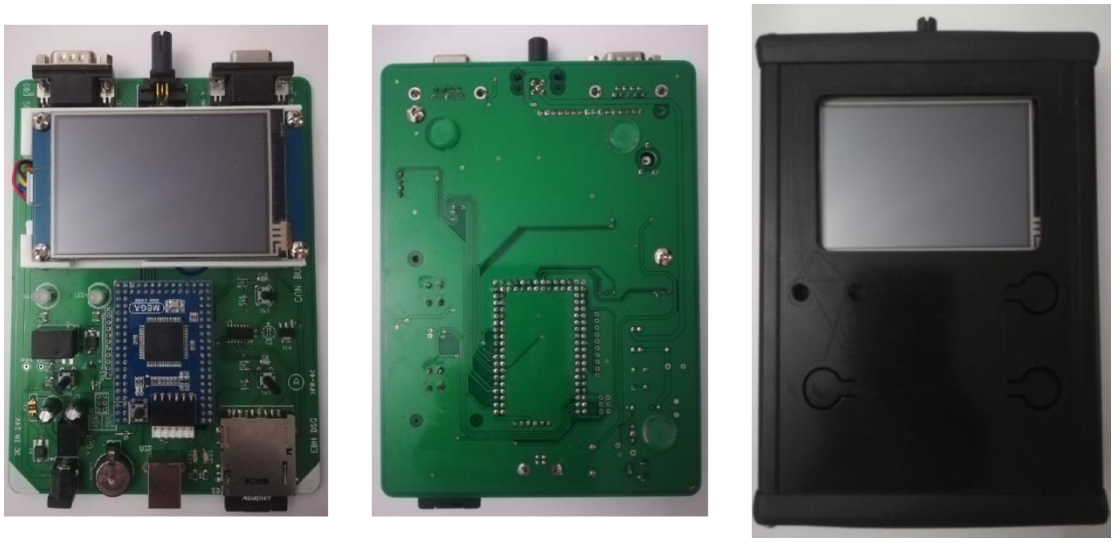


Figura 3-34 Circuito impreso y montaje sobre caja de aluminio.



Especificaciones para la fabricación del PCB:

- 2 Capas (2 layers)
- Material FR4
- Espesor del PCB 1.6mm
- Solder mask green
- Silkscreen white
- Acabado HASL lead free
- 1 oz Cupper (espesor cobre)

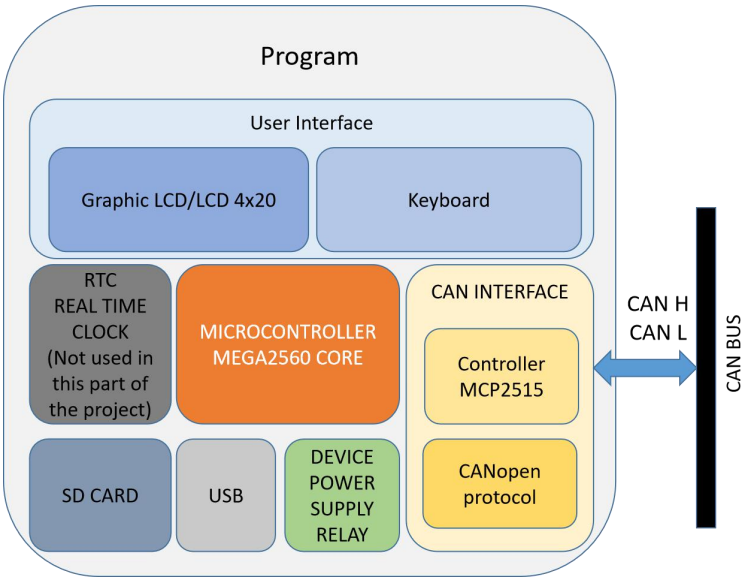
3.8 Diseño del programa para el microcontrolador del equipo.

En la primera parte del desarrollo del proyecto quedaron definidos los códigos CANopen para la configuración de los sensores y en la segunda parte el diseño del Hardware, con los múltiples periféricos a utilizar, en esta tercera parte se describe el desarrollo del programa del microcontrolador, para el control de los puertos de entrada y salida, control de la Interface de usuario e implementación del protocolo CANopen para generación de los códigos para la configuración de los dispositivos CANopen.

El microcontrolador MEGA2560 CORE lleva internamente el bootloader de un Arduino MEGA, la distribución y asignación de las terminales esta asociada a este modulo de desarrollo.

El programa del microcontrolador lo forman los siguientes bloques:

Figura 3-35 Bloques funcionales del programa del microcontrolador.

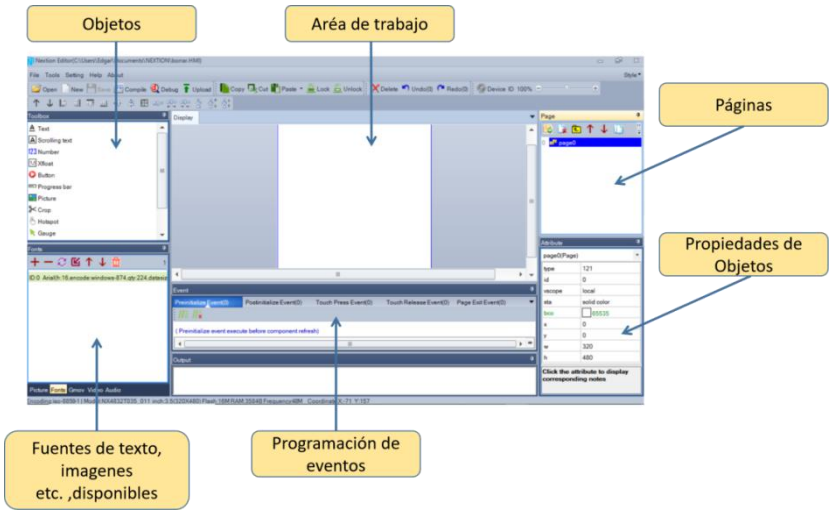


3.8.1 Interfaz gráfica de usuario.

La interfaz de usuario, es el medio por el cual el usuario interactúa con el equipo, de forma visual a través de la pantalla LCD que informa de los eventos y a través del teclado (UP,Down,Enter), para la navegación entre las múltiples opciones del programa.

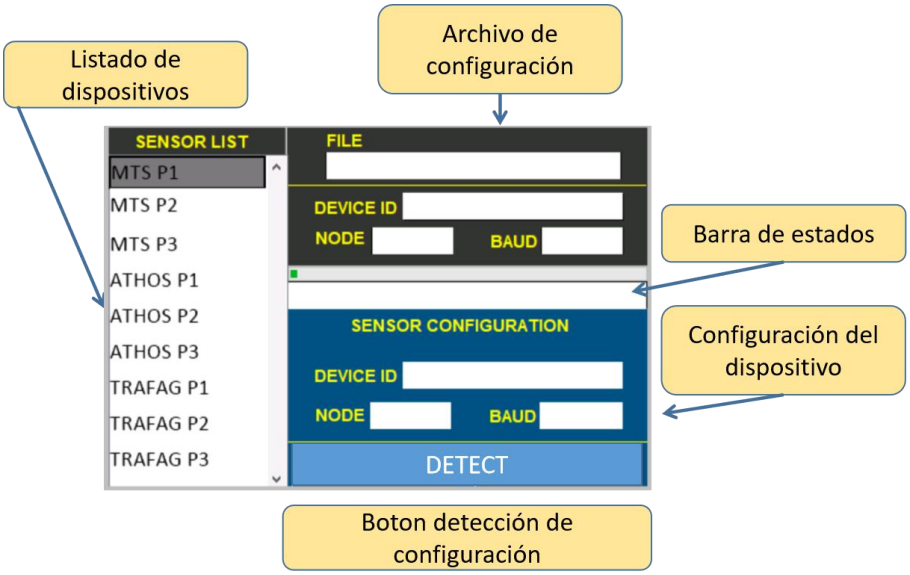
El modulo HMI Nextion NX4832T035 de 3.5”, es un modulo HMI con hardware dedicado, en el que la interfaz gráfica se diseña en un software propio del fabricante, facilitando enormemente el diseño.
El software Nextion, dispone de varias objetos para el diseño de la interfaz, con posibilidad de personalizarlos. Entre los objetos hay pulsadores, barras de desplazamiento, indicadores numéricos y de texto, etc. Cada objeto tiene asociada una referencia, a la que se puede acceder mediante comando UART, por ejemplo un indicador numérico, mediante comunicación UART el microcontrolador envía un comando con el valor que dicho indicador debe mostrar.
El software mostrado tiene 6 áreas, en la que el área central es la de trabajo, donde se agregan objetos desde el listado de la derecha, botones, indicadores etc.
Cada objeto tiene propiedades que son configuradas en el área inferior derecha(propiedades de objetos). Donde se puede configurar las dimensiones, el color, la fuente de letra, alineación del texto etc. A su vez cada objeto puede ser programado para responder a eventos determinados. El área de fuentes de texto, imágenes etc. Es el área donde se incluyen las fuentes de letras, imágenes etc. que se utilizaran en el diseño de la interfaz. En el área de “Paginas” se crean y se accede a paginas de la interfaz para el proyecto.

Figura 3-36 Software nextion.



La interfaz diseñada para la configuración de dispositivos CANopen es la siguiente.

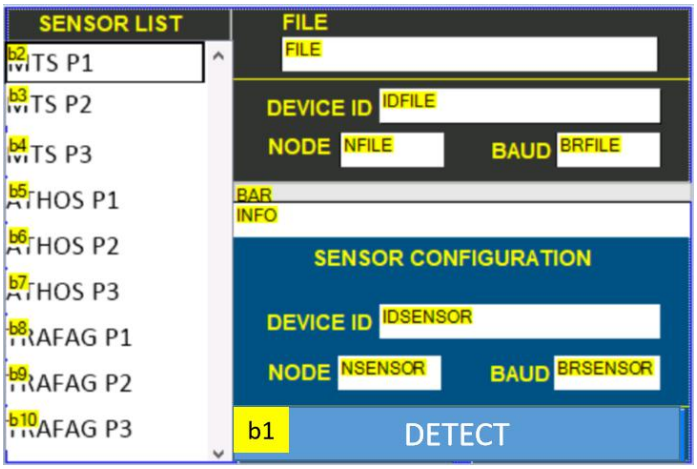
Figura 3-37 Interfaz de configuración de dispositivos CANopen.



Donde en el área de la derecha esta el listado de dispositivos a configurar, MTS, Athos, y Trafag para cada una de las palas. Del listado, una vez seleccionado un dispositivo, la configuración del mismo se muestra en el área “Archivo de configuración” mostrando el Nodo y baud rate. En caso de querer detectar la configuración del sensor conectado se utiliza el botón DETECT, donde una vez solicitada la operación, los datos de la configuración del dispositivo conectado, aparecerán en el área “Configuración del dispositivo”, con el nodo y el baud rate detectados.

Para simplificar el diseño de la interfaz se ha utilizado únicamente pulsadores, aun cuando solo sea para mostrar texto. La asignación de variables para cada objeto en pantalla es:

Figura 3-38 Asignación de variables en la interfaz gráfica.



Los comandos vía UART desde el microcontrolador, se envían en formato ASCII, en minúsculas, y todo comando finalizado con el envió de 3 bytes con valor a 0xFF.
Para el control de la pantalla Nextion se utiliza el puerto serie numero 2 del microcontrolador y los comandos utilizados para enviar texto y cambiar propiedades de color del pulsado son.

Comando para enviar consigna de texto seria

```
Serial2.print(String Var+".txt",String Text);
Serial2.write(0xFF);
Serial2.write(0xFF);
Serial2.write(0xFF);
```

Comando para cambiar el color del pulsador (propiedad de objeto bco).

```
Serial2.print(String "VAR.bco=CODIGO COLOR"
Serial2.write(0xFF);
Serial2.write(0xFF);
Serial2.write(0xFF);
```

Donde VAR es la referencia del objeto cuyas propiedades serán modificadas y CODIGO COLOR es el valor RGB del color a utilizar.

Mediante la modificación de color del pulsador se da el efecto visual que el pulsador ha sido seleccionado. El listado completo del set de instrucciones de Nextion esta en la web en la siguiente pagina: https://nextion.itead.cc/instruction-set/

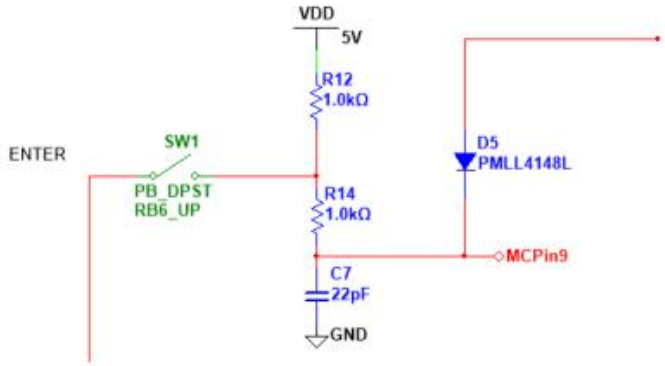
La pantalla Nextion es táctil, pero tiene poca resistencia mecánica y no resulta conveniente para el entorno de trabajo, la pantalla quedara protegida por una caratula transparente e inaccesible al tacto, por lo que dicha característica se deshabilita y la navegación por las diferentes opciones solo es posible a través del teclado.

3.8.2 Control del mando.

Cada push button tiene una función, ENTER, UP y DOWN, asociada a las entradas digitales MCPin 9, MCPin 10, MCPin 11. Los push button del diseño tienen resistencias de pull-up, La enrtada digital es 1 logico(5V) con el push button en OFF, y 0 logico(0 V), con el push button ON . El teclado esta conectado a una interrupción común en la entrada del MCPin 19. La interrupción es utilizada para hacer mas eficiente el programa del teclado, con el fin de evitar supervisar continuamente en un ciclo While() el estado de cada pushbutton. Con la interrupción, el programa mantiene activas sus tareas y cuando la interrupción del teclado es detectada, Interrumpe lo que este haciendo para detectar cual de los 3 push button ha sido presionado.

Uno de los problemas habituales con los push button son los rebotes en la señal que pueden darse, en este proyecto la electrónica cuenta con un circuito antirebotes, que se complementa por software para asegurar que cualquier rebote sea eliminado. Para el sistema antirebote por software, se ha optado por un método simple. Se utilizan un delay(retardo), cuando la interrupción por flanco de bajada es detectada, hay un delay de 2 segundo, tiempo suficiente para que el pulsador haya dejado de ser presionado, lo que asegura que cualquier rebote posterior sea ignorado por la interrupción.

Figura 3-39 Circuito anti rebote por Hardware.



3.8.3 Control del Menú.

Para la navegación entre las diferentes opciones del menú, se utilizan los controles UP y DOWN. Los diferentes botones cambian de color para indicar la posición durante la navegación de los menús. El control ENTER se utiliza para ejecutar la orden, el botón cambia el color del botón por uno de mayor contraste para indicar que ha sido seleccionado.

Todos los botones en la pantalla tienen los colores definidos por defecto en el diseño de la interface. Para dar el efecto de navegación a través de los menús, cada botón tiene tres estados, el estado por defecto, el estado por desplazamiento(UP y DOWM) durante la navegación y el estado por selección cuando ENTER ha sido presionado. Cada estado tiene propiedades de color que se definen en tres Arrange de texto con las instrucciones de Nextion para cambiar el color de cada objeto.
Los tres Arrange son Menu_clr[] que son los valores por defecto, Menu_set[] que son los valores de color por desplazamiento y Menu_ENTER[] que son los colores cuando el boton es presionado. Estos Arrange son declarados como constantes al inicio del programa. La estructura de los Arrange es la siguiente:

Figura 3-40 Estructura de los Arrange para el control de Menús.

```
Menu_clr[0]= "b0.bco=3135";      // Button Back
Menu_clr[1]= "b1.bco=3135";      //DETECT
Menu_clr[2]= "b2.bco=65535";     //MTS 1
Menu_clr[3]= "b3.bco=65535";     //MTS 2
Menu_clr[4]= "b4.bco=65535";     //MTS 3
Menu_clr[5]= "b5.bco=65535";     //Atos 1
Menu_clr[6]= "b6.bco=65535";     //Atos 2
Menu_clr[7]= "b7.bco=65535";     //Atos 3
Menu_clr[8]= "b8.bco=65535";     //Trafag 1
Menu_clr[9]= "b9.bco=65535";     //Trafag 2
Menu_clr[10]= "b10.bco=65535";   //Trafag 3
```

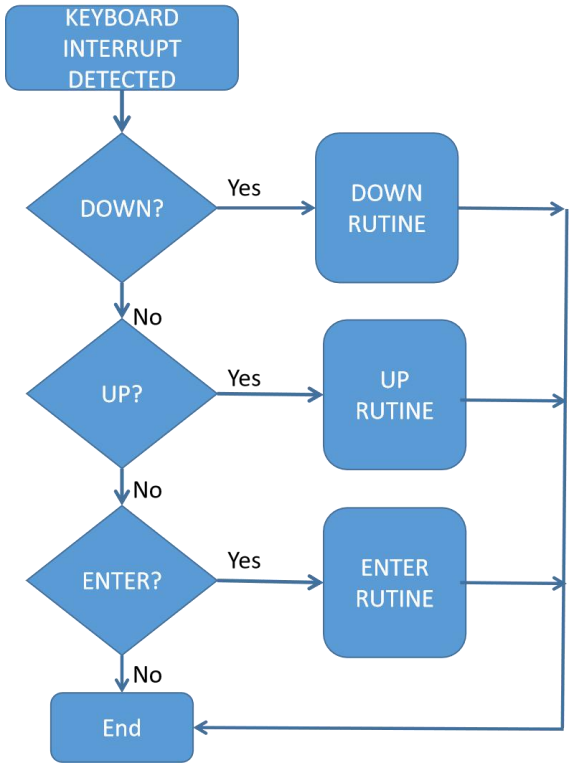
```
Menu_set[0]= "b0.bco=594";       // Button Back
Menu_set[1]= "b1.bco=594";       //DETECT
Menu_set[2]= "b2.bco=31695";     //MTS 1
Menu_set[3]= "b3.bco=31695";     //MTS 2
Menu_set[4]= "b4.bco=31695";     //MTS 3
Menu_set[5]= "b5.bco=31695";     //Atos 1
Menu_set[6]= "b6.bco=31695";     //Atos 2
Menu_set[7]= "b7.bco=31695";     //Atos 3
Menu_set[8]= "b8.bco=31695";     //Trafag 1
Menu_set[9]= "b9.bco=31695";     //Trafag 2
Menu_set[10]= "b10.bco=31695";   //Trafag 3
```

```
Menu_Enter[0]=  "click b0,1";    // Button Back
Menu_Enter[1]=  "click b1,1";    //DETECT
Menu_Enter[2]=  "click b2,1";    //MTS 1
Menu_Enter[3]=  "click b3,1";    //MTS 2
Menu_Enter[4]=  "click b4,1";    //MTS 3
Menu_Enter[5]=  "click b5,1";    //Atos 1
Menu_Enter[6]=  "click b6,1";    //Atos 2
Menu_Enter[7]=  "click b7,1";    //Atos 3
Menu_Enter[8]=  "click b8,1";    //Trafag 1
Menu_Enter[9]=  "click b9,1";    //Trafag 2
Menu_Enter[10]= "click b10,1";   //Trafag 3
```

Ademas de los Arrange se declaran las variables de tipo entero (int) MAXIndex que define el numero máximo de botones, Index que indica la posiciona actual en la navegación.

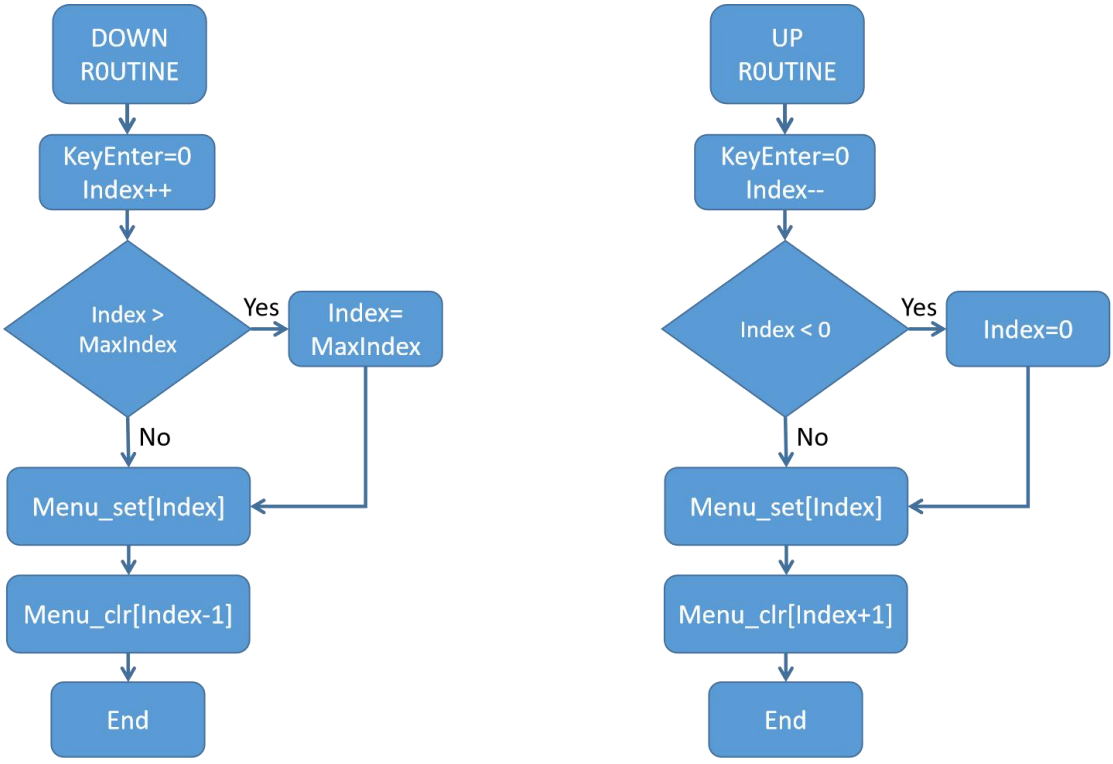
El algoritmo para la navegación es el siguiente:

Figura 3-41 Algoritmo para la navegación en los menús.



Si la interrupción del teclado es detectada el programa detecta cual de los controles ha sido presionado, para ejecutar la rutina que le corresponde a cada control.

Figura 3-42 Algoritmo para el control Up y Down.



Si los controles Down or Up han sido presionados, incrementa/decrementa dentro de los limites[0 - MaxIndex] la posición del cursos, cambiado de color el boton correspondiente a los valores establecidos en el Arrange Menú_set[Index] y devolviendo a su color por defecto al boton seleccionado previamente con los valores establecidos en el Arrange Menu_clr[Index +/- 1]. Logrando dar el efecto visual de la navegación a través de los menús.

Figura 3-43 Efecto visual en la navegación entre los menús.

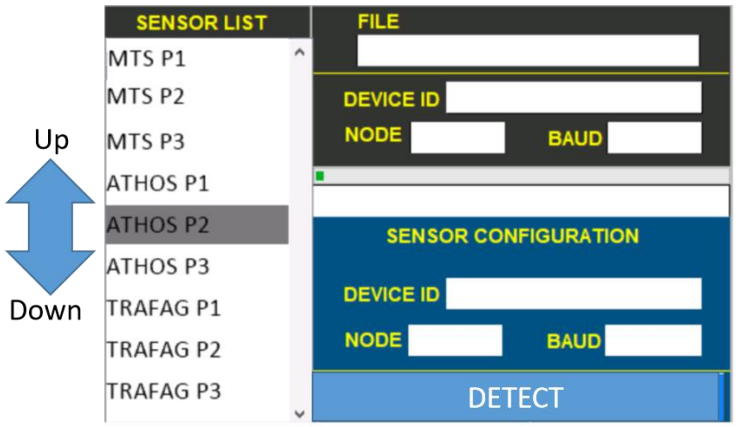
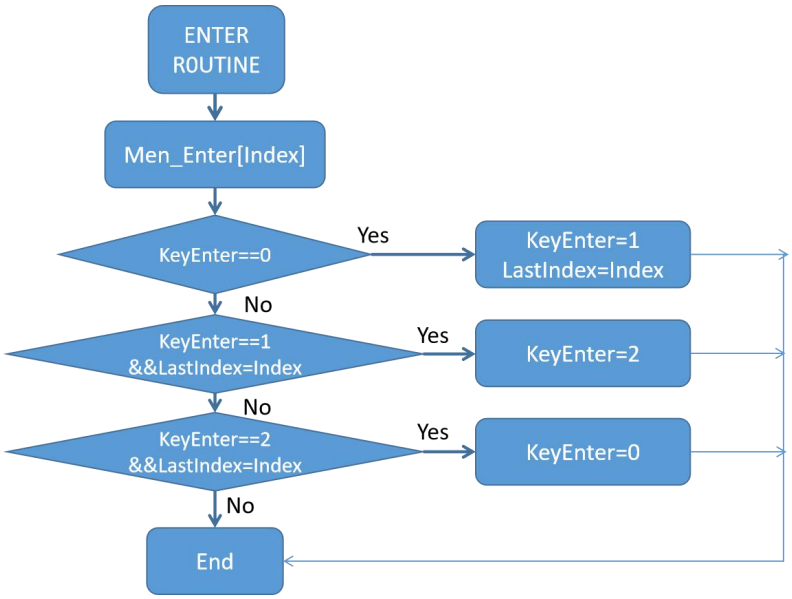


Figura 3-44 Algoritmo para Enter.



El control Enter tiene múltiples funciones, el primer Enter cambia el color del Botón y carga los datos de archivo del sensor a configurar, para mostrarlos en la pantalla, si Enter es presionado de nuevo ejecuta la configuración del sensor y un tercer Enter puede abortar la operación. Por lo que la rutina para la operación de Enter Incrementa la variable KeyEnter si la posición del cursor no ha cambiado.

Como información adicional el teclado puede estar habilitado o deshabilitado con el control de la interrupción.

3.9 Controlador de CAN bus MCP2515.

El MCP2515 es un controlador de CAN, que provee al equipo de la capacidad de transmitir y recibir datos por el CAN bus. La comunicación del controlador CAN con el microcontrolador es por un puerto SPI, a través del cual el microcontrolador configura, controla, envía y recibe datos desde el CAN bus.

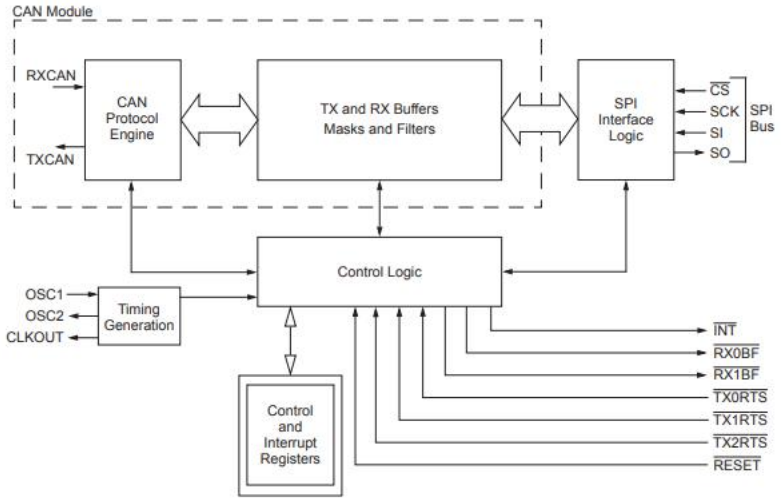
Para el control del MCP2515 existen librerías publicas en el IDE de arduino, pero están diseñadas para ejecutar funciones básicas de transmisión y recepción de mensajes. Y muchas de las características técnicas del controlador CAN MCP2515 de interés para este proyecto no son utilizadas. Por lo que se desarrollan librerías para el control del MCP2515, adaptadas para las necesidades de este proyecto.

Para diseñar las librerías es necesario conocer como esta organizado el dispositivo, conocer los registros y determinar la configuración mas apropiada para los propósitos del proyecto.

3.9.1 Características técnicas del controlador CAN MCP2515.

El MCP2515 internamente esta organizado con la siguiente estructura:

Figura 3-45 Estructura del MCP2515.

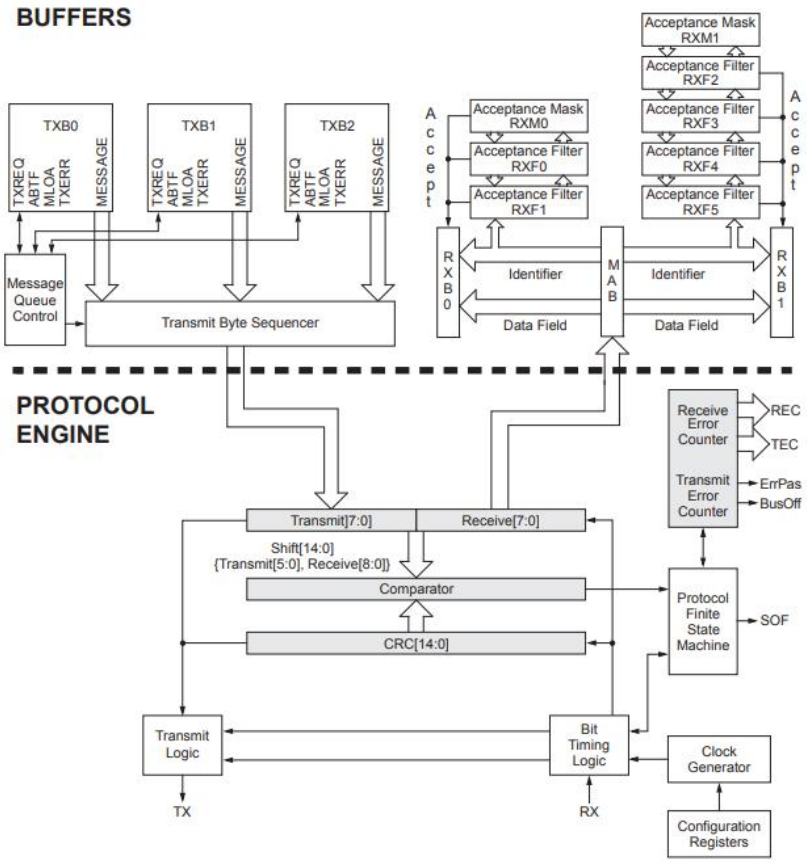


- El modulo CAN, incluye el protocolo para la transmisión y recepción de mensajes CAN, los buffers para la transmisión y recepción de mensajes, configuración de las mascaras y filtros para los buffer de recepción.
- El control lógico contiene los registros que son utilizados para la configuración del dispositivo y definir sus modos de operación.
- El modulo SPI, contiene el protocolo de comunicación para la lectura y escritura sobre los registros del controlador.

El Modulo CAN tiene tres buffers para la transmisión de mensajes y dos buffers para la recepción(Fig. N). De todo el trafico de una red CAN, es posible filtrar cierto tipo de mensajes e ignorar el resto, evitando procesar cada uno de los mensajes del CAN

bus, reduciendo de esta forma la carga de procesamiento del controlador. Esto se logra mediante la configuración de dos Mascaras, una para cada registro y 6 Filtros, dos para el buffer RXB0 y 4 para el RXB1. Existe ademas un buffer intermedio MAB(Message Assembly buffer) donde se cargan los mensajes recibidos desde el Bus CAN,sobreescribiendo siempre sobre el buffer sin importar si previamente había otro mensaje. Si el mensaje recibido es valido y cumple con las condiciones impuestas por los filtros, si el buffer esta libre, carga el mensaje en el buffer correspondiente. Una vez el que alguno de los buffers de recepción(RXB0,RXB1) contiene una trama de comunicación, el microcontrolador debe descargar vía SPI el mensaje de comunicación para vaciar el buffer y permitir que un nuevo mensaje pueda ser alojado, si los buffer de recepción de mensajes no son vaciados, el controlador seguiría cargando mensajes en el MAB pero no podrá cargarlos en los buffers de recepción y los mensajes nuevos recibidos desde el BUS se perderán. Al tener dos buffers de recepción si ambos están configurados para recibir el mismo tipo de mensajes, en caso de que uno de ellos este aun ocupado el nuevo mensaje es cargado en el buffer libre, dando mayor tiempo al microcontrolador de descargar los datos de los dos buffers.

Figura 3-46 Buffers y registros del MCP2515.



3.9.2 Modos de operación.

El controlador CAN tiene diferentes modos de operación.

- **Modo de configuración.** Modo utilizado para la configuración del controlador.

- **Modo normal**. Modo de operación para la transmisión y recepción de tramas de comunicación CAN con los parámetros con los que haya sido configurado el dispositivo.
- **Modo bajo consumo(sleep mode).** Modo en el que el dispositivo mantiene funciones operativas mínimas saliendo de ese estado al detectar actividad en el Bus.
- **Modo de escucha**. Modo en el que el dispositivo, esta deshabilitado para transmitir y solo puede recibir tramas de comunicación desde el bus, pero solo en modo de escucha sin modificar el bit ACK(Acknowledge), que confirma la recepción del mensaje. En este modo el dispositivo conectado puede capturar el trafico del bus, sin interferir entre la comunicación del resto de dispositivos. Este es uno de los modos de operación de interés para este proyecto, puesto que este modo puede ser utilizado para generar un algoritmo de autodetección de baud rate, que es una de las ventajas sobre otros equipos.
- **Modo de bucle.** Este modo es utilizado para depurar el programa, la transmisión de mensajes es interna y dirigida a los buffers de recepción. En este modo la transmisión y recepción de mensajes al BUS CAN esta deshabilitada.

3.9.3 Mapa de registros del controlador CAN.

El controlador CAN cuenta con un conjunto de registros, donde están alojados datos y parámetros de configuración del dispositivo. Los registros y el direccionamiento son de 8 bits, la tabla muestra en colores los registros relativos a los buffers de transmisión y recepción. Y en grises y blancos los relativos a los de control.

Tabla 3-13 Registros del controlador CAN MCP2515.

Lower Address Bits	Higher-Order Address Bits							
	0000 xxxx	0001 xxxx	0010 xxxx	0011 xxxx	0100 xxxx	0101 xxxx	0110 xxxx	0111 xxxx
0000	RXF0SIDH	RXF3SIDH	RXM0SIDH	TXB0CTRL	TXB1CTRL	TXB2CTRL	RXB0CTRL	RXB1CTRL
0001	RXF0SIDL	RXF3SIDL	RXM0SIDL	TXB0SIDH	TXB1SIDH	TXB2SIDH	RXB0SIDH	RXB1SIDH
0010	RXF0EID8	RXF3EID8	RXM0EID8	TXB0SIDL	TXB1SIDL	TXB2SIDL	RXB0SIDL	RXB1SIDL
0011	RXF0EID0	RXF3EID0	RXM0EID0	TXB0EID8	TXB1EID8	TXB2EID8	RXB0EID8	RXB1EID8
0100	RXF1SIDH	RXF4SIDH	RXM1SIDH	TXB0EID0	TXB1EID0	TXB2EID0	RXB0EID0	RXB1EID0
0101	RXF1SIDL	RXF4SIDL	RXM1SIDL	TXB0DLC	TXB1DLC	TXB2DLC	RXB0DLC	RXB1DLC
0110	RXF1EID8	RXF4EID8	RXM1EID8	TXB0D0	TXB1D0	TXB2D0	RXB0D0	RXB1D0
0111	RXF1EID0	RXF4EID0	RXM1EID0	TXB0D1	TXB1D1	TXB2D1	RXB0D1	RXB1D1
1000	RXF2SIDH	RXF5SIDH	CNF3	TXB0D2	TXB1D2	TXB2D2	RXB0D2	RXB1D2
1001	RXF2SIDL	RXF5SIDL	CNF2	TXB0D3	TXB1D3	TXB2D3	RXB0D3	RXB1D3
1010	RXF2EID8	RXF5EID8	CNF1	TXB0D4	TXB1D4	TXB2D4	RXB0D4	RXB1D4
1011	RXF2EID0	RXF5EID0	CANINTE	TXB0D5	TXB1D5	TXB2D5	RXB0D5	RXB1D5
1100	BFPCTRL	TEC	CANINTF	TXB0D6	TXB1D6	TXB2D6	RXB0D6	RXB1D6
1101	TXRTSCTRL	REC	EFLG	TXB0D7	TXB1D7	TXB2D7	RXB0D7	RXB1D7
1110	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT
1111	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL

Note: Shaded register locations indicate that these allow the user to manipulate individual bits using the Bit Modify command.

Filters RXB0 y RXB1 | Mask RXB0 y RXB1 | Buffers TXB0-TXB2 | Buffer RXB0 y RXB1

Los registros en amarillo, contienen la trama de comunicación de los buffer de transmisión TXB0, TXB1 y TXB2. Los registros en Azul contienen los datos de los buffers de recepción RXB0 y RXB1. Estos buffers contienen tramas de comunicación CAN completas, con el identificador en los bytes SIDH,SIDL,EID8,EID0 si el identificador es extendido en los 4 bytes y si es estándar los 11 bits del identificador están contenidos en SIDH, SIDHL. El registro DLC contiene el numero de datos en la trama de comunicación y los datos son los registros D0 a D7.

En verde están los filtros RXF0 a RXF5 y en naranja las mascaras RXM0 a RXM1 aplicados al identificador.

El resto son los registros para la configuración y el control del dispositivo. Que se resumen en la siguiente tabla:

Tabla 3-14 Registros de control del MCP2515.

Register Name	Address (Hex)	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	POR/RST Value
BFPCTRL	0C	—	—	B1BFS	B0BFS	B1BFE	B0BFE	B1BFM	B0BFM	--00 0000
TXRTSCTRL	0D	—	—	B2RTS	B1RTS	B0RTS	B2RTSM	B1RTSM	B0RTSM	--xx x000
CANSTAT	xE	OPMOD2	OPMOD1	OPMOD0	—	ICOD2	ICOD1	ICOD0	—	100- 000-
CANCTRL	xF	REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0	1110 0111
TEC	1C	Transmit Error Counter (TEC)								0000 0000
REC	1D	Receive Error Counter (REC)								0000 0000
CNF3	28	SOF	WAKFIL	—	—	—	PHSEG22	PHSEG21	PHSEG20	00-- -000
CNF2	29	BTLMODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0	0000 0000
CNF1	2A	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	0000 0000
CANINTE	2B	MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE	0000 0000
CANINTF	2C	MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF	0000 0000
EFLG	2D	RX1OVR	RX0OVR	TXBO	TXEP	RXEP	TXWAR	RXWAR	EWARN	0000 0000
TXB0CTRL	30	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-00
TXB1CTRL	40	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-00
TXB2CTRL	50	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-00
RXB0CTRL	60	—	RXM1	RXM0	—	RXRTR	BUKT	BUKT	FILHIT0	-00- 0000
RXB1CTRL	70	—	RSM1	RXM0	—	RXRTR	FILHIT2	FILHIT1	FILHIT0	-00- 0000

BFPCTRL(Dirección 0x0C): Este registro es utilizado para configurar las terminales RX0BF y RX1BF, relativas a las interrupciones que se generan para el microcontrolador cuando alguno de los Buffers RXB0 o RXB1 recibe un mensaje. Las interrupciones permiten que el microcontrolador este liberado de la tarea de supervisar continuamente si hay algún mensaje presente en algunos de los buffers. Con la interrupción habilitada, si el microcontrolador recibe una interrupción, sabe que buffer ha recibido el mensaje y el programa descarga la trama de comunicación CAN recibida.
El registro BFPCTRL tiene estructura mostrada en la tabla siguiente, donde en el recuadro azul se muestran los valores con los que deberá ser configurado el

dispositivo para este proyecto, donde se habilita el uso de la terminales RX0BF y RX1BF y las interrupciones.

Figura 3-47 Registro BFPCTRL, control de interrupciones RXnBF.

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	B1BFS	B0BFS	B1BFE	B0BFE	B1BFM	B0BFM
—	—	0	0	1	1	1	1
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6 **Unimplemented:** Read as '0'

bit 5 **B1BFS:** $\overline{\text{RX1BF}}$ Pin State bit (Digital Output mode only)
- Reads as '0' when RX1BF is configured as interrupt pin

bit 4 **B0BFS:** $\overline{\text{RX0BF}}$ Pin State bit (Digital Output mode only)
- Reads as '0' when RX0BF is configured as interrupt pin

bit 3 **B1BFE:** $\overline{\text{RX1BF}}$ Pin Function Enable bit
1 = Pin function enabled, operation mode determined by B1BFM bit
0 = Pin function disabled, pin goes to high-impedance state

bit 2 **B0BFE:** $\overline{\text{RX0BF}}$ Pin Function Enable bit
1 = Pin function enabled, operation mode determined by B0BFM bit
0 = Pin function disabled, pin goes to high-impedance state

bit 1 **B1BFM:** $\overline{\text{RX1BF}}$ Pin Operation mode bit
1 = Pin is used as interrupt when valid message loaded into RXB1
0 = Digital Output mode

bit 0 **B0BFM:** $\overline{\text{RX0BF}}$ Pin Operation mode bit
1 = Pin is used as interrupt when valid message loaded into RXB0
0 = Digital Output mode

TXRTSCTRL(Direccion 0x0D): Registro de control y configuración de las terminales TXNRTS (TX request to send). Una vez cargadas las tramas de comunicación CAN en algunos de los tres buffers TXBN, hay dos formas de enviar la orden de transmisión, una a través de una orden por el puerto SPI y otra a través de una señal digital en alguna de las 3 terminales TXNRTS. La ultima es la mas eficiente al utilizar menos ciclos de reloj del microcontrolador que enviando una orden vía SPI. El registro TXRTSCTRL permite configurar y supervisar esas terminales. En este proyecto las terminales TX0RTS,TX1RTS y TX2RTS son controlados por las salidas digitales 42,43 y 44 del microcontrolador, por lo que este registro deberá ser configurado al inicio del programa para habilitar esas terminales. Y la estructura del registro y el valor de configuración que deberá tener se muestra en la tabla siguiente.

Figura 3-48 Registro TXRTSCTRL y configuración para el proyecto.

U-0	U-0	R-x	R-x	R-x	R/W-0	R/W-0	R/W-0
—	—	B2RTS	B1RTS	B0RTS	B2RTSM	B1RTSM	B0RTSM
—	—	x	x	x	1	1	1
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6 **Unimplemented:** Read as '0'

bit 5 **B2RTS:** $\overline{\text{TX2RTS}}$ Pin State bit
- Reads state of $\overline{\text{TX2RTS}}$ pin when in Digital Input mode
- Reads as '0' when pin is in 'Request-to-Send' mode

bit 4 **B1RTS:** $\overline{\text{TX1RTX}}$ Pin State bit
- Reads state of $\overline{\text{TX1RTS}}$ pin when in Digital Input mode
- Reads as '0' when pin is in 'Request-to-Send' mode

bit 3 **B0RTS:** $\overline{\text{TX0RTS}}$ Pin State bit
- Reads state of $\overline{\text{TX0RTS}}$ pin when in Digital Input mode
- Reads as '0' when pin is in 'Request-to-Send' mode

bit 2 **B2RTSM:** $\overline{\text{TX2RTS}}$ Pin mode bit
1 = Pin is used to request message transmission of TXB2 buffer (on falling edge)
0 = Digital input

bit 1 **B1RTSM:** $\overline{\text{TX1RTS}}$ Pin mode bit
1 = Pin is used to request message transmission of TXB1 buffer (on falling edge)
0 = Digital input

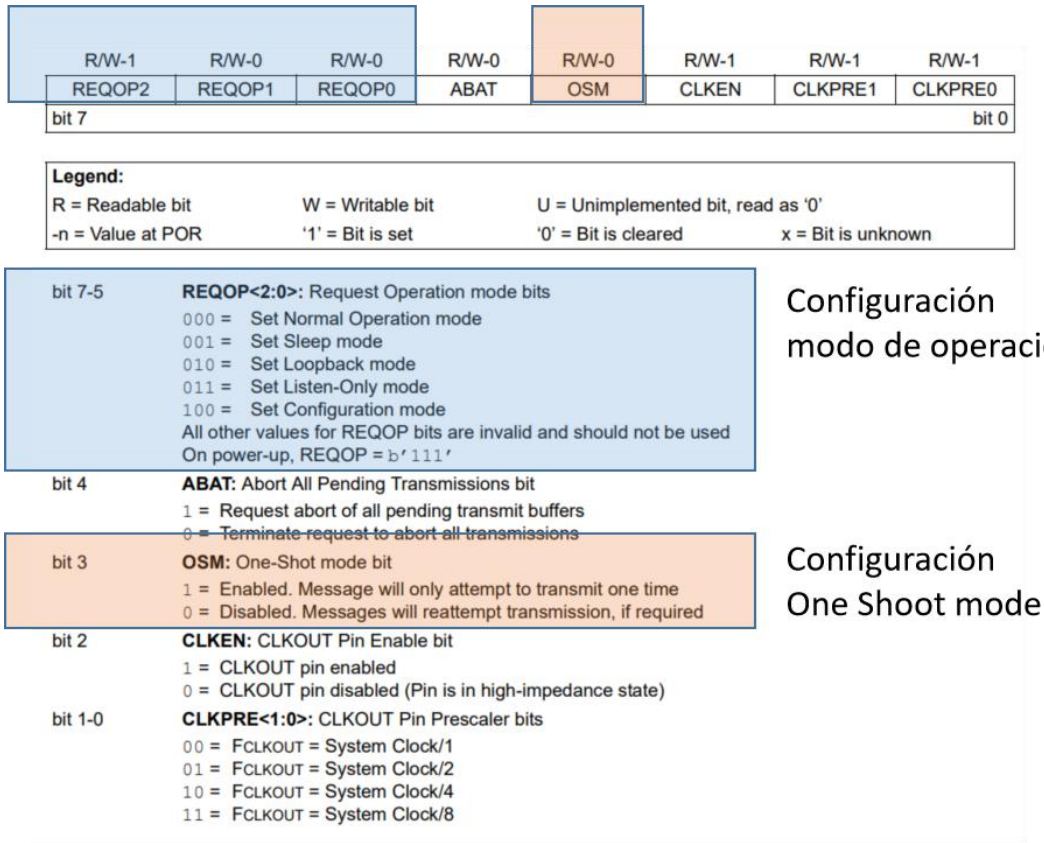
bit 0 **B0RTSM:** $\overline{\text{TX0RTS}}$ Pin mode bit
1 = Pin is used to request message transmission of TXB0 buffer (on falling edge)
0 = Digital input

CANCTRL (Dirección 0xXF). El registro CANCTRL es utilizado principalmente para establecer el modo de operación del dispositivo (Normal, sleep, Loopback, Listen, Configuration) y para la activación de una de las características que resultan interesantes para el proyecto, que es el OSM(One shoot mode). En una transmisión normal el dispositivo se asegura que el mensaje transmitido haya sido recibido, esto es a través del bit ACK, que modifica el receptor para indicar al transmisor que el mensaje ha sido recibido. En una comunicación normal si el bit ACK no es conmutado por el receptor o si se detecta un error en la transmisión, el transmisor entiende que el mensaje no ha sido recibido y volverá a retransmitir el mensaje continuamente hasta que el receptor reciba el mensaje. Habilitando el bit OSM del registro CANCTRL, el mensaje se transmite una sola vez en el CAN bus, sin importar si el mensaje ha sido recibido o si ha habido un error en la transmisión. El resto de los bits son utilizados para habilitar una terminal de reloj de salida y configurar un preescaldor para la frecuencia de reloj. Función que no es utilizada en este proyecto.

CANSTAT(Dirección 0xXE): Este registro es de solo lectura y guarda relación con el registro anterior. Es utilizado para saber el modo de operación actual del dispositivo y de las interrupciones que están activas.

Ambos registros se muestran en las tablas siguientes:

Figura 3-49 Registro de control CANCTRL.



R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-1
REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0
bit 7							bit 0

Legend:

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-5 **REQOP<2:0>:** Request Operation mode bits
000 = Set Normal Operation mode
001 = Set Sleep mode
010 = Set Loopback mode
011 = Set Listen-Only mode
100 = Set Configuration mode
All other values for REQOP bits are invalid and should not be used
On power-up, REQOP = b'111'

Configuración modo de operación

bit 4 **ABAT:** Abort All Pending Transmissions bit
1 = Request abort of all pending transmit buffers
0 = Terminate request to abort all transmissions

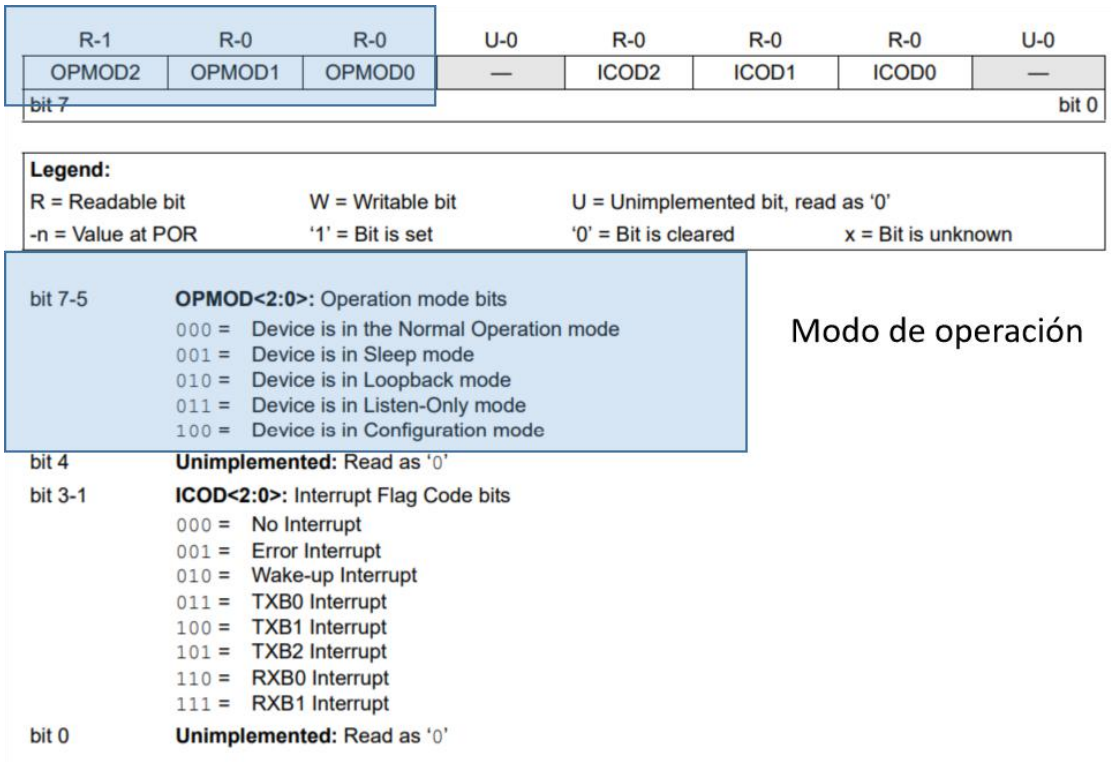
bit 3 **OSM:** One-Shot mode bit
1 = Enabled. Message will only attempt to transmit one time
0 = Disabled. Messages will reattempt transmission, if required

Configuración One Shoot mode

bit 2 **CLKEN:** CLKOUT Pin Enable bit
1 = CLKOUT pin enabled
0 = CLKOUT pin disabled (Pin is in high-impedance state)

bit 1-0 **CLKPRE<1:0>:** CLKOUT Pin Prescaler bits
00 = FCLKOUT = System Clock/1
01 = FCLKOUT = System Clock/2
10 = FCLKOUT = System Clock/4
11 = FCLKOUT = System Clock/8

Figura 3-50 Registro de estados CANSTAT.



R-1	R-0	R-0	U-0	R-0	R-0	R-0	U-0
OPMOD2	OPMOD1	OPMOD0	—	ICOD2	ICOD1	ICOD0	—
bit 7							bit 0

Legend:

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-5 **OPMOD<2:0>:** Operation mode bits
000 = Device is in the Normal Operation mode
001 = Device is in Sleep mode
010 = Device is in Loopback mode
011 = Device is in Listen-Only mode
100 = Device is in Configuration mode

Modo de operación

bit 4 **Unimplemented:** Read as '0'

bit 3-1 **ICOD<2:0>:** Interrupt Flag Code bits
000 = No Interrupt
001 = Error Interrupt
010 = Wake-up Interrupt
011 = TXB0 Interrupt
100 = TXB1 Interrupt
101 = TXB2 Interrupt
110 = RXB0 Interrupt
111 = RXB1 Interrupt

bit 0 **Unimplemented:** Read as '0'

TEC(0x1C)(Transmit Error counter) y REC(0x2D)(Receive error counter): El protocolo CAN tiene sofisticados mecanismos de detección de errores. El MCP2515 contiene dos contadores de 8 bits(TEC y REC), que registran los errores en las operaciones de transmisión y recepción de mensajes CAN. Cuando cualquiera de los dos contadores superan los 128 errores el controlador CAN entra en el modo BUS OFF temporalmente. Que impide la recepción y transmisión de mensajes. Relacionado con la detección de errores esta el registro siguiente.

EFLG (0x2D) (Error Flag register). Este es un registro de 8 bits, en el que cada bit indica un estado de error, que puede ser Overflow, cuando cuando se recibe un mensaje nuevo y los buffers RXBN están aun ocupados. O cuando alguno de los contadores REC y TEC superan los 128 errores. En el caso de este proyecto no se han utilizado estos registros y se mencionan brevemente.

CNF1,CNF2,CNF3 (Configuration register). Todos los nodos conectados a una red CAN deben estar configurados con el mismo baud rate. En el controlador MCP2515 los registros CNF1,2,3 son utilizados para configurar el baud rate del controlador. Hay una serie de ecuaciones y procedimientos complejos para calcular los valores de estos registros. Pero existen herramientas Online que con la velocidad del cristal, el tipo de controlador CAN y el baud rate deseado, calcula los valores de estos registros. La herramienta utilizada para este proyecto es:
https://www.kvaser.com/support/calculators/bit-timing-calculator/

Los valores de los registros para los diferentes baud rates soportados son:

Tabla 3-15 Configuración del baud rate.

Configuracion de los registros CNFn para el baud rate			
Baud rate (kb/s)	CNF1	CNF2	CNF3
10	0x04	0xB6	0x27
20	0x04	0xB6	0x13
50	0x03	0xAC	0x09
100	0x03	0xAC	0x04
125	0x01	0x91	0x07
250	0x03	0xAC	0x01
500	0x03	0xAC	0x00
800	0x01	0x9A	0x00
1000	0x01	0x91	0x00

CANINTE(0x2B) Interrupt Enable. EL MCP2515 tiene 8 fuentes de interrupción, el registro CANINTE es el registro que debe ser configurado para habilitar las interrupciones, la tabla siguiente muestra la estructura del registro y el listado de interrupciones posibles..

Figura 3-51 Registro para habilitación de interrupciones CANINTE.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **MERRE:** Message Error Interrupt Enable bit
1 = Interrupt on error during message reception or transmission
0 = Disabled

bit 6 **WAKIE:** Wake-up Interrupt Enable bit
1 = Interrupt on CAN bus activity
0 = Disabled

bit 5 **ERRIE:** Error Interrupt Enable bit (multiple sources in EFLG register)
1 = Interrupt on EFLG error condition change
0 = Disabled

bit 4 **TX2IE:** Transmit Buffer 2 Empty Interrupt Enable bit
1 = Interrupt on TXB2 becoming empty
0 = Disabled

bit 3 **TX1IE:** Transmit Buffer 1 Empty Interrupt Enable bit
1 = Interrupt on TXB1 becoming empty
0 = Disabled

bit 2 **TX0IE:** Transmit Buffer 0 Empty Interrupt Enable bit
1 = Interrupt on TXB0 becoming empty
0 = Disabled

bit 1 **RX1IE:** Receive Buffer 1 Full Interrupt Enable bit
1 = Interrupt when message received in RXB1
0 = Disabled

bit 0 **RX0IE:** Receive Buffer 0 Full Interrupt Enable bit
1 = Interrupt when message received in RXB0
0 = Disabled

Para el proyecto las interrupciones utilizadas y habilitadas son RX0E y RX1E, interrupciones generadas que indican que hay un mensaje nuevo en el Buffer RXB0 o RXB1 y las interrupciones TX0E, TX1E,TX2E activas. Después de una orden de transmisión de mensaje (RTS request to send) dicha interrupción es a través del pin 12 (INT) del controlador MCP2515. El resto de interrupciones no están habilitadas de momento para este proyecto. Por lo que el valor de configuración para este registro es: b00011111 (0x1F).

CANINTF(0x2C) CAN intrrupt flag. Este registro esta relacionado con el registro anterior que es utilizado para habilitar las interrupciones durante la configuración del dispositivo. El CANINTF es utilizado para indicar las interrupciones activas. Una vez activa, debe ser reiniciada por el microcontrolador, para borrar la interrupción y permitir que pueda generarse una nueva. La estructura del registro es la siguiente:

Figura 3-52 Registro CANINTF CAN interrupt flag.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **MERRF:** Message Error Interrupt Flag bit
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

bit 6 **WAKIF:** Wake-up Interrupt Flag bit
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

bit 5 **ERRIF:** Error Interrupt Flag bit (multiple sources in EFLG register)
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

bit 4 **TX2IF:** Transmit Buffer 2 Empty Interrupt Flag bit
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

bit 3 **TX1IF:** Transmit Buffer 1 Empty Interrupt Flag bit
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

bit 2 **TX0IF:** Transmit Buffer 0 Empty Interrupt Flag bit
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

bit 1 **RX1IF:** Receive Buffer 1 Full Interrupt Flag bit
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

bit 0 **RX0IF:** Receive Buffer 0 Full Interrupt Flag bit
1 = Interrupt pending (must be cleared by MCU to reset interrupt condition)
0 = No interrupt pending

TXBnCTRL (0x30,0x40,0x50) Transmit buffer control. Este registro es de Control/Supervisión relacionada con los buffers de transmisión. Tiene varias funciones. De la tabla que describe el registro, el microcontrolador escribe vía comando SPI un 1 en el bit TXREQ, para enviar una orden de transmisión del contenido del Buffer TXBn (Si el controlador esta configurado para recibir la orden de transmisión a través de las entradas digitales TXnRTS hace exactamente la misma función). Una vez realizada la orden de transmisión, el microcontrolador puede supervisar el mismo bit para saber si el mensaje ha sido transmitido. Si el mensaje ha sido transmitido su valor vuelve a 0, si su valor sigue siendo 1 significa que el mensaje aun esta pendiente de ser transmitido y los bits MLOA o TXERR, pueden indicar la razón. MLOA si el mensaje a perdido la arbitraje de acceso al bus y TXERR si ha habido un error de transmisión a través del BUS. Los bits TXP1 y TXP0, son utilizados para configurar la prioridad del mensaje. Así si los tres buffers llevan orden de transmisión simultanea, el primer buffer transmitido, sera el configurado con mayor prioridad. Para los algoritmos de configuración CAN se utilizara únicamente el buffer de transmisión TXB0, por lo que la configuración en el orden e prioridad es intrascendente. La estructura del registro se muestra en la tala siguiente.

Figura 3-53 TXBnCTRL Transmision buffer control.

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

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **Unimplemented:** Read as '0'

bit 6 **ABTF:** Message Aborted Flag bit
1 = Message was aborted
0 = Message completed transmission successfully

bit 5 **MLOA:** Message Lost Arbitration bit
1 = Message lost arbitration while being sent
0 = Message did not lose arbitration while being sent

bit 4 **TXERR:** Transmission Error Detected bit
1 = A bus error occurred while the message was being transmitted
0 = No bus error occurred while the message was being transmitted

bit 3 **TXREQ:** Message Transmit Request bit
1 = Buffer is currently pending transmission
(MCU sets this bit to request message be transmitted - bit is automatically cleared when the message is sent)
0 = Buffer is not currently pending transmission
(MCU can clear this bit to request a message abort)

bit 2 **Unimplemented:** Read as '0'

bit 1-0 **TXP<1:0>:** Transmit Buffer Priority bits
11 = Highest Message Priority
10 = High Intermediate Message Priority
01 = Low Intermediate Message Priority
00 = Lowest Message Priority

RXB0CTRL (0x60), RXB1CTRL(0x70h) Receive buffer control. Estos registro de configuración y control son para el buffer RXB0 y RXB1. Donde algunos de los bits son utilizados para configurar el funcionamiento del buffer y otros utilizados por el control para extraer información adicional del tipo de mensaje recibido.
Para los algoritmos de configuración de sensores CANopen, interesa configurar estos registros, para recibir cualquier tipo de mensaje(extendido o estándar) sin las restricciones impuestas por los filtros. De la tabla N, Los bits RXM1 y RXM0 del registro, serán configurados para inhabilitar los filtros y recibir cualquier tipo de mensaje (Configuración b11). El resto de los bits son bits para control, que dan información de la correspondencia del mensaje recibido con el numero de filtro configurado, al estar inhabilitados los Filtros estos bits no son utilizados. El valor para la configuración inicial de ambos registros sera **0x60**. Con los filtros deshabilitados, el buffer RXB0 sera el primer buffer al que sea redirigido un mensaje recibido en el buffer temporal MAB, en caso de estar ocupado por que el microcontrolador no ha descargado el buffer, el mensaje sera alojado en el buffer RXB1, evitando que se pierda nuevo mensaje recibido, dando un margen de tiempo adicional para que el microcontrolador descargue y vacíe los dos buffers ocupados.

Figura 3-54 RXB0CTRL Registro de control Buffer RXB0.

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

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **Unimplemented:** Read as '0'

bit 6-5 **RXM<1:0>:** Receive Buffer Operating mode bits
- 11 = Turn mask/filters off; receive any message
- 10 = Receive only valid messages with extended identifiers that meet filter criteria
- 01 = Receive only valid messages with standard identifiers that meet filter criteria. Extended ID filter registers **RXFnEID8:RXFnEID0** are ignored for the messages with standard IDs.
- 00 = Receive all valid messages using either standard or extended identifiers that meet filter criteria. Extended ID filter registers **RXFnEID8:RXFnEID0** are applied to first two bytes of data in the messages with standard IDs.

bit 4 **Unimplemented:** Read as '0'

bit 3 **RXRTR:** Received Remote Transfer Request bit
- 1 = Remote Transfer Request Received
- 0 = No Remote Transfer Request Received

bit 2 **BUKT:** Rollover Enable bit
- 1 = RXB0 message will rollover and be written to RXB1 if RXB0 is full
- 0 = Rollover disabled

bit 1 **BUKT1:** Read-only Copy of BUKT bit (used internally by the MCP2515)

bit 0 **FILHIT0:** Filter Hit bit – indicates which acceptance filter enabled reception of message
- 1 = Acceptance Filter 1 (RXF1)
- 0 = Acceptance Filter 0 (RXF0)

Note: If a rollover from RXB0 to RXB1 occurs, the FILHIT bit will reflect the filter that accepted the message that rolled over.

Figura 3-55 RXB1CTRL registro de control Buffer RXB.

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

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **Unimplemented:** Read as '0'

bit 6-5 **RXM<1:0>:** Receive Buffer Operating mode bits
- 11 = Turn mask/filters off; receive any message
- 10 = Receive only valid messages with extended identifiers that meet filter criteria
- 01 = Receive only valid messages with standard identifiers that meet filter criteria
- 00 = Receive all valid messages using either standard or extended identifiers that meet filter criteria

bit 4 **Unimplemented:** Read as '0'

bit 3 **RXRTR:** Received Remote Transfer Request bit
- 1 = Remote Transfer Request Received
- 0 = No Remote Transfer Request Received

bit 2-0 **FILHIT<2:0>:** Filter Hit bits - indicates which acceptance filter enabled reception of message
- 101 = Acceptance Filter 5 (RXF5)
- 100 = Acceptance Filter 4 (RXF4)
- 011 = Acceptance Filter 3 (RXF3)
- 010 = Acceptance Filter 2 (RXF2)
- 001 = Acceptance Filter 1 (RXF1) (Only if BUKT bit set in RXB0CTRL)
- 000 = Acceptance Filter 0 (RXF0) (Only if BUKT bit set in RXB0CTRL)

En resumen los valores para la configuración inicial del dispositivo sera el de la siguiente tabla.

Tabla 3-16 Configuración de lo registros de control del MCP2515.

											CONFIGURACION	
Register Name	Address (Hex)	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	POR/RS Value	BIN	HEX
BFPCTRL	0C	—	—	B1BFS	B0BFS	B1BFE	B0BFE	B1BFM	B0BFM	--00 00	--00 1111	0x0F
TXRTSCTRL	0D	—	—	B2RTS	B1RTS	B0RTS	B2RTSM	B1RTSM	B0RTSM	--xx x0	--XX 0111	0x07
CANSTAT	xE	OPMOD2	OPMOD1	OPMOD0	—	ICOD2	ICOD1	ICOD0	—	100- 00	---- ----	-----
CANCTRL	xF	REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0	1110 01	*OM	
TEC	1C	Transmit Error Counter (TEC)								0000 00	---- ----	-----
REC	1D	Receive Error Counter (REC)								0000 00	---- ----	-----
CNF3	28	SOF	WAKFIL	—	—	—	PHSEG22	PHSEG21	PHSEG20	00-- -0	*BR	
CNF2	29	BTLMODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0	0000 00	*BR	
CNF1	2A	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	0000 00	*BR	
CANINTE	2B	MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE	0000 00	0001 1111	0x1F
CANINTF	2C	MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF	0000 00	---- ----	-----
EFLG	2D	RX1OVR	RX0OVR	TXBO	TXEP	RXEP	TXWAR	RXWAR	EWARN	0000 00	---- ----	-----
TXB0CTRL	30	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-	---- ----	-----
TXB1CTRL	40	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-	---- ----	-----
TXB2CTRL	50	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-	*T	
RXB0CTRL	60	—	RXM1	RXM0	—	RXRTR	BUKT	BUKT	FILHIT0	-00- 00	0110 0000	0x60
RXB1CTRL	70	—	RSM1	RXM0	—	RXRTR	FILHIT2	FILHIT1	FILHIT0	-00- 00	0110 0000	0x60

*OM. Los valores dependen del modo de operación configurado durante el programa del microcontrolador.

*BR Los valores dependen del valor del baud rate configurad por el microcontrolador.

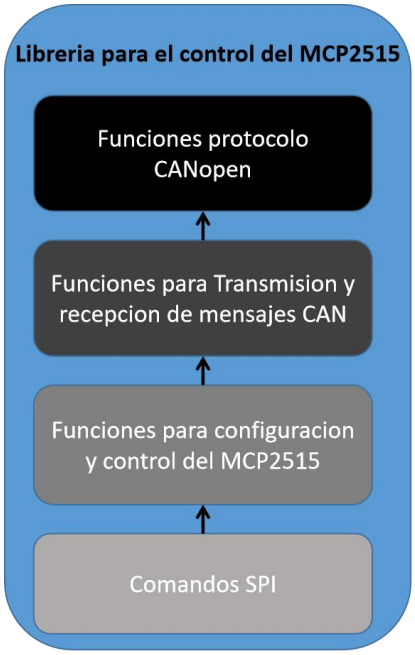
*T Los valores dependen de los asignados durante la transmisión de los datos.

Los valores mostrados en la tabla son los utilizados para la configuración del controlador MCP2515, parámetros ajustados a las necesidades de los algoritmos para configurar los sensores CANopen.

3.10 Diseño de las librerías para el controlador CAN MCP2515.

Las librerías CAN diseñadas para el proyecto están basadas en los comandos SPI del MCP2515, que son operaciones básicas para leer y escribir sobre los registros del controlador. A partir de esta base se construyen nuevas funciones mas complejas. Las funciones e la librería CAN desarrollada la podríamos dividir en 4 de acuerdo a la Figura siguiente.

Figura 3-56 Estructura de la librería CAN desarrollada para e proyecto.



Comandos SPI: Conjunto de comandos SPI del controlador para leer y escribir sobre los registros y efectuar tareas de supervisión y control.

Funciones para configuración, supervision y control del MCP2515: Combinación de comandos SPI, para realizar operaciones de configuración, supervisión y control.

Funciones Tx,Rx de CAN bus: Combinación de funciones de para la transmisión y recepción de tramas de comunicación CAN.

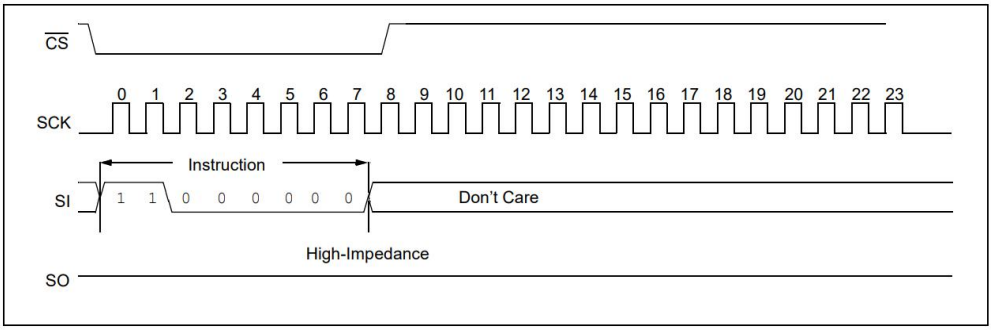
Funciones protocolo CANopen: Funciones para implementación de operaciones del protocolo CANopen.

Las hojas de especificaciones del MCP2515 definen un conjunto de comandos para supervisión, control, leer y escribir sobre los registros del controlador. La utilización de los comandos SPI se describen a continuación.

3.10.1 Comando RESET.

El comando RESET es utilizada para reiniciar el controlador con todos los registros con sus valores por defecto. Después de reiniciado el dispositivo entra al modo de configuración. Este comando es equivalente al Reset por hardware en el pin 17

Figura 3-57 Comando RESET(xC0)



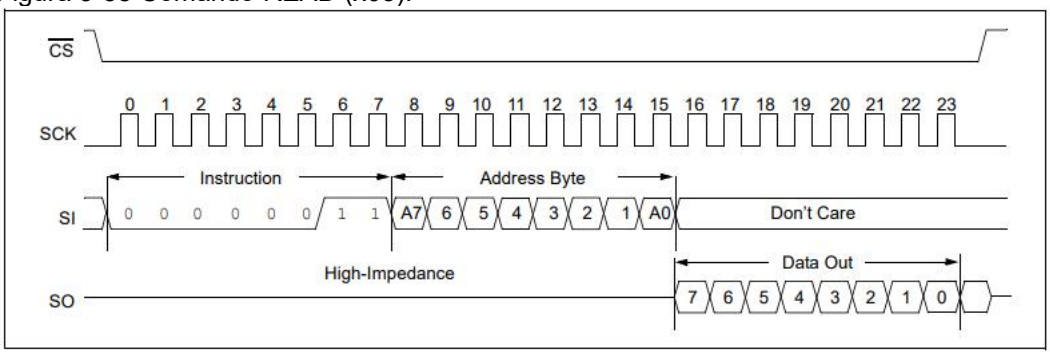
Para la librería del controlador la función para el comando es la siguiente:

```
void CAN::RESET(void)
{
   digitalWrite(CS, LOW);          //CS LOW    Start SPI communication
   SPI.transfer(SPI_RESET);        //SPI transfer RESET Command
   digitalWrite(CS, HIGH);         //CS HIGH Stop SPI communication
   delay(5);                       //Delay to the normal operation after RESET
}
```

3.10.2 Comando READ.

El comando READ (b00000011) es utilizada para leer los valores de uno o mas registros. Del diagrama de tiempos, el flanco de bajada del Pin CS(Chip select) inicia la comunicación SPI, a partir de la cual el Microcontrolador por la linea SI(Slave in) envía el comando READ (b0000 0011) seguido de la dirección de 8 bits (A7-A0) del registro a leer. A continuación, los datos almacenados en el registro con la dirección especificada son enviados al microcontrolador por la linea SO(Slave otput). Una vez el dato ha sido transmitido, el puntero interno de la dirección se incrementa automáticamente cada 8 bits. Lo que hace posible leer consecutivamente a partir de una dirección inicial, los datos de los registros siguientes, manteniendo la señal de sincronía SCK hasta que el que Pin CS tenga un flanco de subida. Mediante la lectura de registros consecutivos se evita tener que reenviar el comando READ para cada uno de los registros leídos.

Figura 3-58 Comando READ (x03).



Para este comando se generaron dos funciones en la librería para el MCP2515.

CAN.READ Que permite leer el contenido de un solo registro en la dirección "Address" que es el parámetro de entrada y con valor de retorno el dato del registro.

```
byte CAN::READ(byte Address)
{
   byte Data;                          //Var declaration
```

```
  digitalWrite(CS, LOW);          //CS LOW Start SPI communication
  SPI.transfer(SPI_READ);         //SPI tranfer READ command
  SPI.transfer(Address);          //SPI transfer Address
  Data = SPI.transfer(0x00);                //SPI READ Data
  digitalWrite(CS, HIGH);         //CS HIGH Stop SPI communication
  return Data;                    //Return Data
}
```

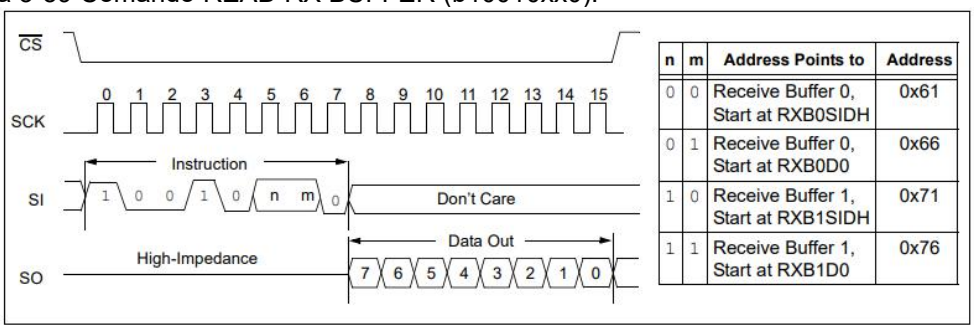
CAN.READ_M Función que permite leer consecutivamente datos de varios registros, desde una dirección inicial "Address", hasta un numero determinado de registros "Ndatos". Retornando a través de una apuntador un array "AData", con los valores de los registros recibidos.

```
void CAN::READ_M(byte Address, byte *AData, byte Ndatos)
{
  digitalWrite(CS, LOW);          //CS LOW Start SPI communication
  SPI.transfer(SPI_READ);         //SPI transfer READ command
  SPI.transfer(Address);          //SPI start address transfer
  for (byte i = 0; i < Ndatos; i++)       //Transfer Ndatos
  { AData[i] = SPI.transfer(0x00); }

  digitalWrite(CS, HIGH);         //CS HIGH Stop SPI communication
}
```

3.10.3 Comando READ RX BUFFER.

La instrucción READ RX BUFFER proporciona un medio para leer rápidamente un buffer de recepción (RXB0 o RXB1). Esta instrucción reduce la sobrecarga del puerto SPI en un byte (El byte de dirección). Los valores n y m del byte de comando determinan la posición del puntero a una dirección inicial de acuerdo a la tabla del diagrama de tiempos, pudiendo acceder al buffer desde la dirección del COBID o directamente desde la dirección de los datos. Al igual que la instrucción READ, el comando permite leer consecutivamente los registros siguientes hasta que el pin CS cambia a nivel alto. El comando simplifica aun mas la sobrecarga del puerto SPI al borrar automáticamente la interrupción asociada al buffer (CANINTF.RXnIF) cuando CS para a nivel alto, finalizando el comando.

Figura 3-59 Comando READ RX BUFFER (b10010xx0).



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

La función para esta librería es:

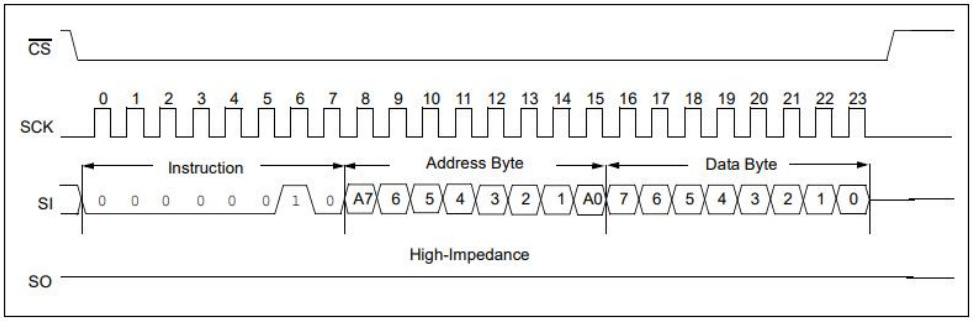
CAN.READ_RX_BUFFER. La función que recibe como parámetros, la dirección del buffer se quiere leer (RXB0 o RXB1), el apuntador del array donde se guardaran los datos recuperados del buffer "RXData" y el numero de datos que se espera leer "Ndata".

```
void CAN::READ_RX_BUFFER(byte Address, byte *RxData, byte Ndata)
{
   byte ReadRxBuffer;

   switch (Address) {                    //Get command
      case RXB0SIDH:          ReadRxBuffer = SPI_READ_RX | 0x0; break;
      case RXB0D0:            ReadRxBuffer = SPI_READ_RX | 0x2; break;
      case RXB1SIDH:          ReadRxBuffer = SPI_READ_RX | 0x4; break;
      case RXB1D0:            ReadRxBuffer = SPI_READ_RX | 0x6; break;
      default:                ReadRxBuffer = SPI_READ_RX | 0x0; break;
   }

   digitalWrite(CS, LOW);                //CS LOW start SPI communication

   SPI.transfer(ReadRxBuffer);           //SPI tranfer READRXBUFFER command
 // Gen. clock signal to receive data over MISO
 for (byte i = 0; i < Ndata; i++)
   { RxData[i] = SPI.transfer(0x00); }
   digitalWrite(CS, HIGH);               //STOP SPI Communication
}
```

3.10.4 Comando BYTE WRITE.

El comando WRITE es utilizado para escribir sobre los registros. Del diagrama de tiempo, con el flanco de bajada CS, se envía la instrucción WRITE, seguida de la dirección(Address) y un byte de datos. Una vez completada la operación, el puntero interno de la dirección apunta a la siguiente dirección por lo que es posible escribir en registros secuenciales siempre que el nivel de CS sea bajo y la señal de sincronía SCK se mantenga.

Figura 3-60 Comando WRITE (x02).



Para la librería se generan dos funciones para este comando.

- **CAN.WRITE.** Función que escribe sobre un solo registro con al dirección "Address", un byte de datos "Data.

```
void CAN::WRITE(byte Address, byte Data)
{
digitalWrite(CS, LOW);          //CS LOW to start SPI Communication
SPI.transfer(SPI_WRITE);        //SPI transfer WRITE Command
SPI.transfer(Address);          //SPI transfer Address
SPI.transfer(Data);             //SPI transfer Data
digitalWrite(CS, HIGH);         // CS HIGH stop SPI communication
}
```

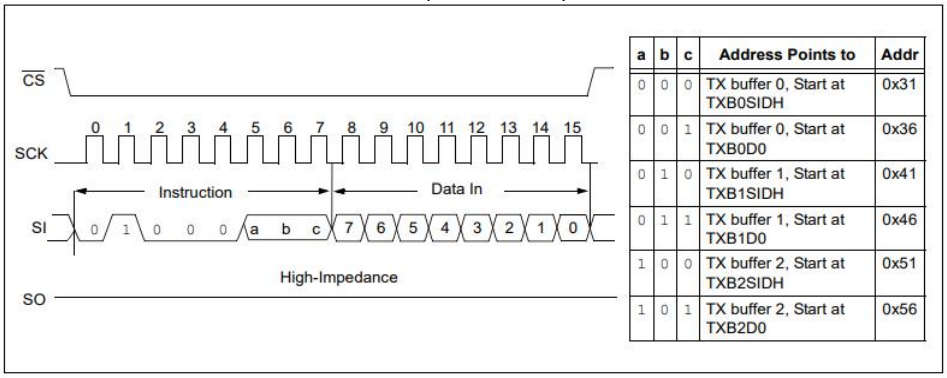
- **CAN.WRITE_M.** Funcion adicional que permite escribir secuencialmente los datos del array DataArray[], empezando por la dirección "Address", hasta la dirección del registro "Address" + "Ndata".

```
void CAN::WRITE_M(byte Address, byte DataArray[], byte Ndata)
{
  digitalWrite(CS, LOW);                  //CS LOW, start SPI communication
  SPI.transfer(SPI_WRITE);                //SPI transfer WRITE command
  SPI.transfer(Address);                  //SPI transfer start Address
  for (byte i = 0; i < Ndata; i++)        //Write Data from Address
  {SPI.transfer(DataArray[i]); }          // Until Address + Ndata
  digitalWrite(CS, HIGH);                 //CS HIGH stop SPI communication
}
```

3.10.5 Comando LOAD TX BUFFER.

El comando LOAD TX BUFFER simplifica la operación de escribir sobre los buffers de Transmisión TXB0,TXB1 y TXB2. A diferencia del comando WRITE la dirección "Address" no es requerida. En los tres bits a,b,c del mismo comando se apunta a una de las 6 direcciones posibles, a partir de las cuales, se empieza a escribir. Los apuntadores pueden empezar a partir de la dirección del COB-ID, o directamente sobre los datos de cualquiera de los tres buffers de transmisión(TXB0,TXB1,TXB2).

Figura 3-61 Comando LOAD TX BUFFER (b01000xxx).



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

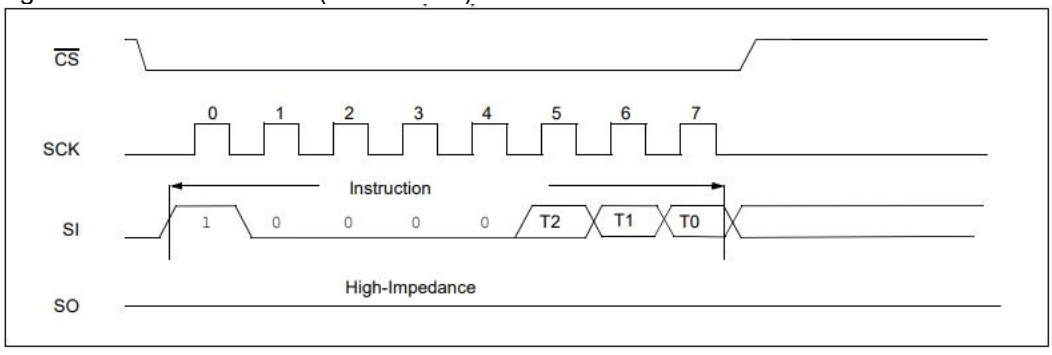
Para la librería, la función generada para este comando es:

```
void CAN::LOAD_TX(byte Address, byte Data)
{
   byte LoadTxBuffer;

   switch (Address) {                          //check Start point address
      case TXB0SIDH:     LoadTxBuffer = SPI_WRITE_TX | 0x0; break;
      case TXB0D0:        LoadTxBuffer = SPI_WRITE_TX | 0x1; break;
      case TXB1SIDH:     LoadTxBuffer = SPI_WRITE_TX | 0x2; break;
      case TXB1D0:        LoadTxBuffer = SPI_WRITE_TX | 0x3; break;
      case TXB2SIDH:     LoadTxBuffer = SPI_WRITE_TX | 0x4; break;
      case TXB2D0:        LoadTxBuffer = SPI_WRITE_TX | 0x5; break;
      default:            LoadTxBuffer = SPI_WRITE_TX | 0x0; break;
   }

   digitalWrite(CS, LOW);                      //CS LOW Sstart SPI communication
   SPI.transfer(LoadTxBuffer);                 //SPI transfer LOAD command
   SPI.transfer(Data);                         //SPI transfer DATA
   digitalWrite(CS, HIGH);                     //CS HIGH Strop SPI communication
}
```

3.10.6 Comando RTS(Request to send).

El comando RTS se utiliza para enviar una orden de transmisión para uno o mas de los buffersTXBn. Después del flanco de bajada de CS, se envía el comando RTS a través de la linea SI(Slave Input). Donde los bits T2,T1,T0, indican los buffer habilitados para transmitir. El comando RTS, habilita automáticamente para aquellos buffers a transmitir el bit control TXBnCTRL.TXR.

Figura 3-62 Comando RTS(b1000 0xxx).



Para las librerías del controlador MCP2515 se ha creado una función para el comando vía SPI. El comando es equivalente a una señal digital en los pines TXnRTS, cuya solicitud es mas rápida que a través del comando SPI. Aun así con el objetivo de completar el set de instrucciones vía SPI, se crea la función siguiente.

```
void CAN::RTS(byte TXBN)
{   byte RTSTxn;
   switch (TXBN) {                  // Adjust command for TXBn
      case TXB_0:       RTSTxn = SPI_RTS | 0x1; break;
      case TXB_1:       RTSTxn = SPI_RTS | 0x2; break;
      case TXB_2:       RTSTxn = SPI_RTS | 0x4; break;
      default:          RTSTxn = SPI_RTS | 0x1; break;
   }
   digitalWrite(CS, LOW);       //CS LOW start SPI com
   SPI.transfer(RTSTxn);        //SPI transfer RTS TXn
   digitalWrite(CS, HIGH);       //CS HIGH Stop SPI com
}
```

3.10.7 Comando BIT MODIFY.

El comando BIT MODIFY permite escribir sobre bits individuales en un registro específico de control o de estado. El comando no es valido para todos los registros, del mapa de registros de la Figura N, solo los registros sombreados lo permiten.

En el flanco de bajada del CS se inicia la comunicación SPI enviando el microcontrolador a través de la linea SI(Slave input), el comando BIT MODIFY seguido de la dirección(Address byte) del registro, la Mascara(MASK byte) y los Datos.

El MASK byte determina que bits en el registro serán modificados como se muestra en la figura siguiente.

Figura 3-63 Ejemplo modificación de bits sobre un registro.

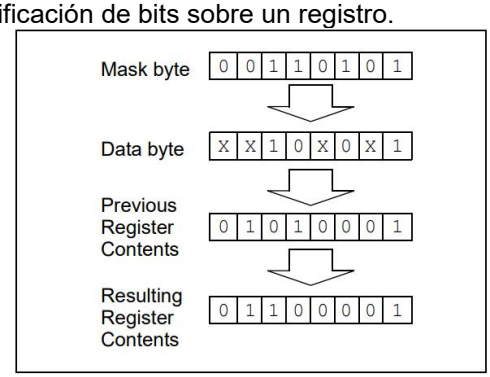
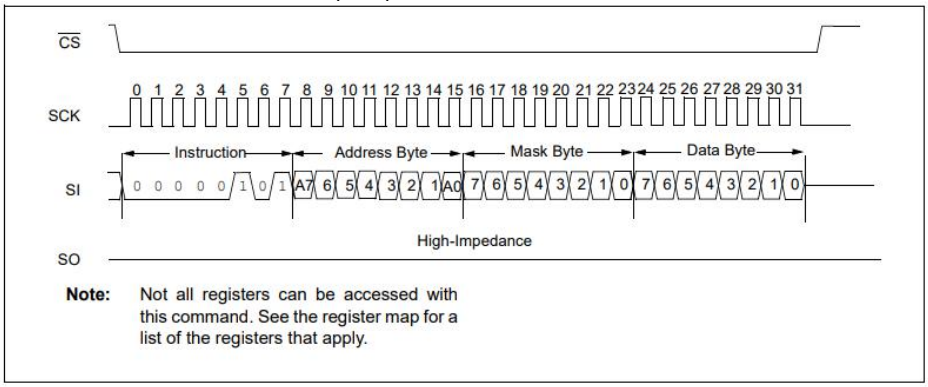


Figura 3-64 Comando BIT MODIFY(x05).



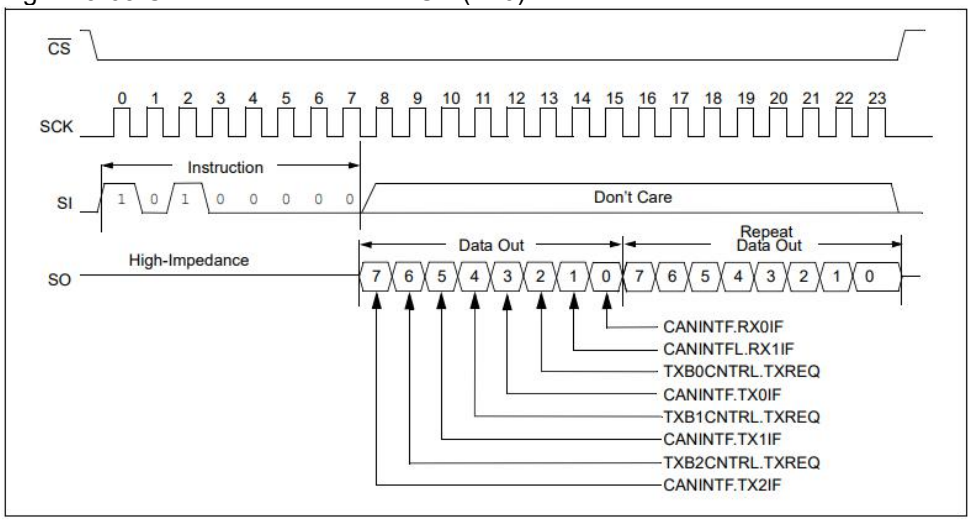
Para la librería la función para el comando BYTE MODIFY es la siguiente, recibe como parámetros la dirección del registro “Address”, la Mascara “MASK” y el dato “Data”.

```
void CAN::BIT_MODIFY(byte Address, byte Mask, byte Data)
{
digitalWrite(CS, LOW);                  //CS bit low to start SPI transfer command

SPI.transfer(SPI_BIT_MODIFY);                   //SPI transfer BIT MODIFY COMMAND
b00000101    SPI.transfer(Address);                         //SPI transfer Address
SPI.transfer(Mask);                             //SPI transfer MASK
SPI.transfer(Data);                     //SPI tranfer DATA

digitalWrite(CS, HIGH);                 //CS bit HIGH END SPI Command
}
```

3.10.8 Comando READ STATUS.

El comando READ STATUS permite a través de un solo comando, el acceso a algunos de los bits de estado utilizados con frecuencia para supervisión de recepción y transmisión de mensajes. Con el flanco de bajada del CS, se envía el comando READ STATUS por el puerto SI(Slave Input), respondiendo el MCP2515 al microcontrolador por el puerto SO(Slave otput) con un dato de 8 bits, donde cada bit corresponde a un estado como se muestra en el diagrama de tiempos.

Figura 3-65 Comando READ STATUS (xA0).



La función creada para este comando, devuelve el byte de estados y el programa es el siguiente:

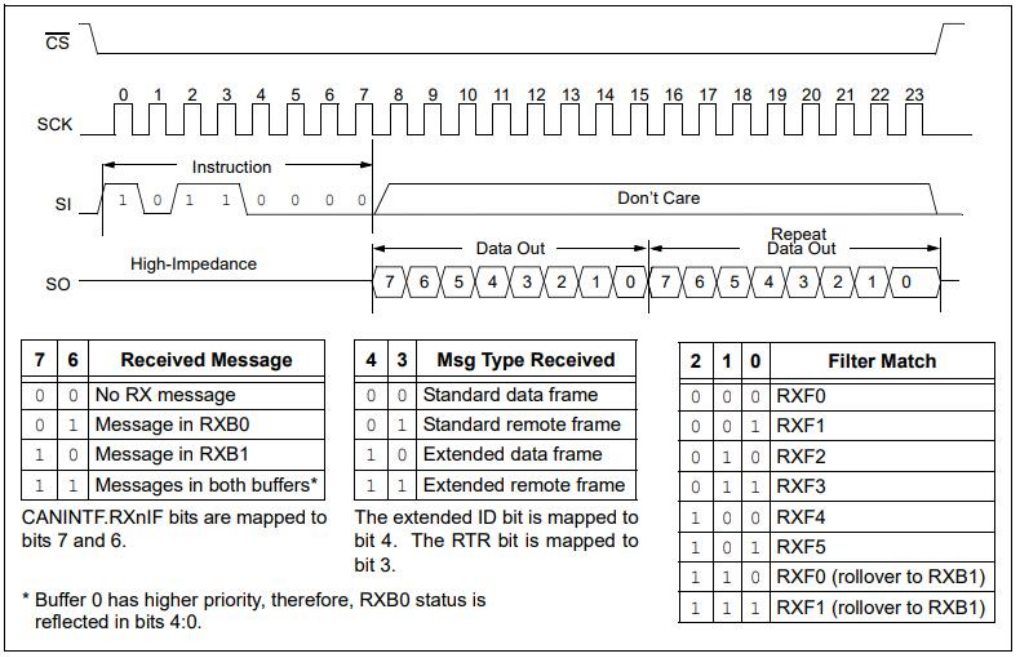
```
byte CAN::READ_STATUS(void)
{
byte Status;                            //Declaration Variable Status
digitalWrite(CS, LOW);                  //CS LOW to start SPI communication
```

```
SPI.transfer(SPI_READ_STATUS);      //SPI Command READ STATUS b1010 0000
Status = SPI.transfer(0x00);        //SPI READ STATUS DATA
digitalWrite(CS, HIGH);             //CS HIGH STOP SPI Communication
return Status;                      //Return Status Data
}
```

3.10.9 Comando RX STATUS.

RX STATUS se usa para supervisar el estado de los buffers RXBn y determinar rápidamente si un mensaje ha sido recibido, en que buffer(RXB0 o RXB1), el tipo de mensaje (Extendido o estándar) y el filtro coincidente con el mensaje recibido.

Figura 3-66 Comando RX STATUS (xB0).



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

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

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

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

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

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

Para la librería la función desarrollada para este comando es la siguiente.

```
byte CAN::RX_STATUS(void)
{
  byte Status;                      //Var declaration
  digitalWrite(CS, LOW);            //CS LOW to start SPI communicaction
  SPI.transfer(SPI_RX_STATUS);      //SPI Transfer RX STATUS command
  Status = SPI.transfer(0x00);      //Read Status
  digitalWrite(CS, HIGH);           //CS HIGH to stop SPI communication
  return Status;                    // Return Status
}
```

Con estos códigos quedan cubiertos los comandos SPI del controlador a partir de los cuales se construyen funciones mas complejas. Se lista a continuación un Resumen de los comandos SPI.

Resumen comandos SPI.

```
void CAN::RESET(void)
byte CAN::READ(byte Address)
void CAN::READ_M(byte Address, byte *AData, byte Ndatos)
void CAN::READ_RX_BUFFER(byte Address, byte *RxData, byte Ndata)
void CAN::WRITE(byte Address, byte Data)
void CAN::WRITE_M(byte Address, byte DataArray[], byte Ndata)
void CAN::LOAD_TX(byte Address, byte Data)
void CAN::RTS(byte TXBN)
void CAN::BIT_MODIFY(byte Address, byte Mask, byte Data)
byte CAN::READ_STATUS(void)
byte CAN::RX_STATUS(void)
```

3.10.10 Configuración, supervisión y control del MCP2515.

El listado de funciones esta diseñadas para facilitar la configuración de los registros mas importantes y para realizar tareas de supervisión y control de los registros de transmisión y recepción del MCP2515.

3.10.11 SETMODE.

Función utilizada para cambiar el modo de operación, actúa sobre el registro CANCTRL, bits REQOP<2:0>. Y utiliza el comando BIT_MODIFY para modificar solo los bits relativos al modo de operación. La función recibe un array de caracteres con el modo de operación a configurar, “NORMAL”, “SLEEP”, “LOOPBACK”, ”LISTEN”, ”CONFIGURATION”, cada array esta asociado a su valor binario en el registro, utilizando una mascara 0xE, con el comando BIT_MODIFY, modifica únicamente los bits asociados la modo de operación del controlador. La funciones es la siguiente.

```
void CAN::SETMODE(char MODE[])
{
   byte ROMode;         //Request Operation mode
   if (MODE == "NORMAL")                  {ROMode = 0x00;}
   else if (MODE == "SLEEP")              {ROMode = 0x20;}
   else if (MODE == "LOOPBACK")           {ROMode = 0x40;}
   else if    (MODE == "LISTEN")          {ROMode = 0x60;}
   else if    (MODE == "CONFIGURATION")   {ROMode = 0x80;}
   else                                   { ROMode = 0x00;}

   BIT_MODIFY(CANCTRL, 0xE0, ROMode);
}
```

3.10.12 BAUDCONFIG.

Esta función configura el baud rate del controlador, escribiendo sobre los registros CNF3,CNF2,CNF1, los valores correspondientes para un Baud rate determinado. La función recibe como parámetro el Baud rate, que lo asocia a valores de los registros CNFn. Mediante el comando WRITE_M se escribe sobre los tres registros consecutivos. La función es la siguiente.

```
void CAN::SPEEDCONFIG(int BAUDRATE)
{
   byte conf[3];

   // array configspeed{CNF3,CNF2,CNF1}    switch (BAUDRATE) {
      case 10:       conf[0] = 0x04; conf[1] = 0xB6; conf[2] = 0x27;          break;
      case 20:       conf[0] = 0x04; conf[1] = 0xb6; conf[2] = 0x13;          break;
      case 50:       conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x09;          break;
      case 100:      conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x04;          break;
      case 125:      conf[0] = 0x01; conf[1] = 0x91; conf[2] = 0x07;          break;
      case 250:      conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x01;          break;
      case 500:      conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x00;          break;
      case 800:      conf[0] = 0x01; conf[1] = 0x9A; conf[2] = 0x00;          break;
      case 1000:     conf[0] = 0x01; conf[1] = 0x91; conf[2] = 0x00;          break;
   }

   WRITE_M(CNF3, conf, 3);
}
```

3.10.13 STATUSRXDATA.

Función para supervisar el estado de los buffers RXB0 y RXB1. La función utiliza el comando RX_STATUS y separa en tres variables su contenido, para facilitar su análisis.

```
void CAN::STATUSRXDATA(byte *RxBuffer, byte *MsgType, byte *FilterMatch)
{
   byte Status;

   Status = RX_STATUS();

   *RxBuffer = (Status & 0xC0) >> 6; //get the bits related with received message
   *MsgType = (Status & 0x18) >> 3;
   *FilterMatch = (Status & 0x07);
}
```

3.10.14 CLEARBUFFERRXN.

Función utilizada para limpiar los bits asociados a la notificación de recepción de mensajes del registro CANINTF. Una vez descargado el buffer RXBn el bit RXnIF es

puesto a cero para permitir que el buffer RXBn reciba nuevos mensajes. Esta funcion utiliza el comando BIT_MODIFY para escribir únicamente sobre los bits relativos a RXnIF.

```
void CAN::CLEARBUFFERRXN(byte RXN)
{
   byte MASK;
   if(RXN==0) MASK=0x01;
   if(RXN==1) MASK=0x02;
   BIT_MODIFY(CANINTF, MASK, 0x00);
}
```

3.10.15 RXBNCTRL_CONFIG.

Función utilizada para configurar los registros RXBnCTRL, donde son habilitados los filtros y definidos el tipo de mensajes que pueden recibir los buffers RXB0 y RXB1 (Estandar o extendido).Los registros pueden ser configurados diferentes entre si, para simplificar la operación ambos buffers se configuran iguales. Para el caso del Buffer RXB0, la función rollover se configura con la misma función. Esta función utiliza el comando MODIFY_BIT, para modificar únicamente los bits que interesa configurar.

```
void CAN::RXBNCTRL_CONFIG(char RX_MODE[], char ROLLOVER_ENABLE)
{
  byte RXM;         //RXmode value    byte RXRE;       //RX Rollover enable bit    byte
  RXBN_CONFIG = 0;
   if (RX_MODE == "ANYM")         {RXM = B11;}    //RX any message,Filters OFF
   else if (RX_MODE == "EXTM")    {RXM = B10;}    // RX extended only, Filters ON
   else if (RX_MODE == "STDM")    {RXM = B01;}    //RX standart only, Filters ON
   else if (RX_MODE == "ALLVM")    {RXM = B00;}    //RX STD and EXT,Filters ON

//Config Rollover bit
   if (ROLLOVER_ENABLE == 'E') {RXRE = 0X01;    } //Rollover Enable
   else if (ROLLOVER_ENABLE == 'D') {RXRE = 0x00; } //Rollover disbale

//Build RXB0CTRL and RXB1CTRL
   RXM = RXM << 5;
   RXRE = RXRE << 3;
   RXBN_CONFIG = RXM & RXRE;

//Write over RXB0CTRL and RXB1CTRL
   BIT_MODIFY(RXB0CTRL, 0x64, RXBN_CONFIG); // MASK 0x64 B01100100
   RXBN_CONFIG = RXM;
   BIT_MODIFY(RXB1CTRL, 0x60, RXBN_CONFIG);
}
```

3.10.16 SETMASK_RXMN.

Esta función configura la mascara para los filtros, se utiliza el comando WRITE_M para escribir sobre los registros RXM0SIDH, RXM0SIDL, RXM1SIDH y RXM1SIDL. La función esta implementada solo para mensajes estándar y solo funciona cuando el MCP2515 esta en modo de configuración.

```
void CAN::SETMASK_RXMN(byte RXMASKN, unsigned int MASK_HEX)
{
   byte MASK_ID[2];

   MASK_ID[0] = byte ((MASK_HEX & 0x7F8) >> 3);
   MASK_ID[1] = byte ((MASK_HEX & 0x007) << 5);
   //Write over RXMnSIDH and RXMnSIDL register at same time.
   WRITE_M(RXMASKN, MASK_ID,2);
}
```

3.10.17 SETFILTER_RXFN.

Esta función es utilizada para configurar los filtros individualmente. Recibe como parámetros, la dirección del filtro RXFnSIDH y el valor del filtro. Utiliza el comando WRITE_M para escribir desde RXFnSIDH hasta RXFnSIDL del filtro correspondiente, el valor del filtro de 2 bytes. Solo es posible escribir sobre los filtros y las mascaras si el controlador MCP2515 esta en modo de configuración. El código de la función se muestra a continuación.

```
void CAN::SETFILTER_RXFN(byte RXFILTERN, unsigned int FILTER_HEX)
  {
     byte FILTER_ID[2];

     FILTER_ID[0]=byte ((FILTER_HEX & 0x7F8) >> 3);      //GET SIDH
     FILTER_ID[1]=byte ((FILTER_HEX & 0x007) << 5);      //GET SIDL

 //Write over RXMnSIDH and RXMnSIDL register at same time.
     WRITE_M(RXFILTERN,FILTER_ID,2);
   }
```

Resumen Funciones configuración, supervisión y control.

```
void CAN::SETMODE(char MODE[])
void CAN::SPEEDCONFIG(int BAUDRATE)
void CAN::STATUSRXDATA(byte *RxBuffer, byte *MsgType, byte *FilterMatch)
void CAN::CLEARBUFFERRXN(byte RXN)
void CAN::RXBNCTRL_CONFIG(char RX_MODE[], char ROLLOVER_ENABLE)
void CAN::SETMASK_RXMN(byte RXMASKN, unsigned int MASK_HEX)
void CAN::SETFILTER_RXFN(byte RXFILTERN, unsigned int FILTER_HEX)
```

3.11 Funciones para recepción y transmisión de mensajes CAN.

El siguiente conjunto de funciones son de mas alto nivel que las anteriores están destinadas a completar el objetivo de las librerías que es transmitir y recibir mensajes CAN.

Para las librerías de alto nivel, se define un tipo de variable para las tramas de comunicación CAN de tipo Struct, que contiene los diferentes elementos de una trama de comunicación CAN.
El tipo de variable StdFrame tiene la siguiente estructura:

```
typedef struct {
unsigned int     COB_ID;        //11 bits Data length
byte RTR_BIT;                   //Requeste to Remote 1 bit
byte DLC;                       // Data length 4 bit Data    value range 0- 8
byte DATA[8];                   // Data 0-8 bytes
 } StdFrame;
```

Para transmitir un mensaje CAN, este primero debe ser cargado en alguno de los buffers TXBn y posteriormente enviar un comando SPI de transmisión RTS o a través de la señal digital TXnRTS. La función para cargar el buffer con un dato es la siguiente.

3.11.1 LOADTX.

Esta función carga un buffer TXBn con una trama de comunicación CAN en formato StdFrame, la estructura descrita anteriormente. La función descompone los elementos de StdFrame, en una array de valores de 8 bits, para ser cargados en los registros desde TXBnSIDH hasta TXBnDm. La función recibe los parámetros TXBN que es la dirección del registro TXBnSIDH y la trama de comunicación CAN en formato StdFrame. La función es la siguiente.

```
void CAN::LOADTX(byte TXBN, StdFrame CanMsg)
{
byte TXAddress, IDH, IDL, DLCbyte;    //Priority value
byte DTX[14] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00}; // TXBNCTRL,SIDH, SIDL, EID8 (0x00),EID0(0x00),DLC, TXBnDM

IDH = byte((CanMsg.COB_ID & 0x7F8) >> 3);          //Get byte SIDH from COB_ID
IDL = byte((CanMsg.COB_ID & 0x007) << 5);          //Get byte SIDL from COB_ID
DLCbyte = ( CanMsg.RTR_BIT    << 6 ) | CanMsg.DLC; //Get DLC byte

TXAddress = TXBN; //TXBN;
DTX[1] = IDH;    //SIDH
DTX[2] = IDL;    //SIDL
DTX[3] = 0x00; //EID8
DTX[4] = 0x00; //EID0
```

```
DTX[5] = DLCbyte;

  for (byte i = 0; i < CanMsg.DLC; i++)      //Get Data
  {
     DTX[6 + i] = CanMsg.DATA[i];
  }

  for (byte i = 0; i < 14; i++)              //Load buffer TXBn with DTX[]
  {
     WRITE(TXAddress, DTX[i]);
     TXAddress++;
  }
  }
```

3.11.2 SENDTXBn.

Esta funcionó utiliza el comando RTS para transmitir el buffer solicitado, adicionalmente la función comprueba que el buffer ha sido enviado supervisando el bit TXBnCTRL.TXREQ que indica si el mensaje ha sido transmitido (0) o si aun esta pendiente de transmitir. Valor que devuelve la función indicando que la transmisión ha sido efectuada correctamente

```
byte CAN::SENDTXBn(byte TXBN)
{
  byte TX_Transmit;        //0 Send TXBN sucesfull      1 Fail to send TXBN     byte
  TXSTATUS, i = 0;

  RTS(TXBN);               //SPI command RTS

  delay(10);               //wait 10 ms to check if the message has been transmited

  while (TXSTATUS == 1)       //Check bit TXBnCTRL.TXREQ
  {
     TXSTATUS = (READ(TXBN) & 0x78) >> 3;
     i++;
     if (i == 100) break;
  }

  if (i == 100)    TX_Transmit = 1;          // if transmir fail return TX_Transmit 1
  else TX_Transmit = 0;                      // if transmit sucesfull TX_Transmit =0
  return (TX_Transmit);                      //Return Transmit status
}
```

Para la recepción de mensajes CAN, consiste en supervisar, si alguno de los dos buffers de recepción ha recibido un mensaje nuevo. Una vez confirmado el

microcontrolador debe descargar el buffer y reiniciar la interrupción para que el buffer pueda recibir nuevos mensajes. Se han implementado dos métodos para este fin, mediante una funcion que supervisa continuamente el estado de los buffers y mediante las interrupciones RXB0INT y RXB1INT.

3.11.3 WAITRXN.

Función para la supervision continua del estado de los buffers RXBn, la función utiliza el comando STATUSRX, para supervisar el estado de los buffers durante un tiempo, si no han sido recibidos mensajes la función devuelve 0, pero si los buffers reciben algun nuevo mensaje detecta en cual de los dos buffers (pueden ser ambos) y los descarga. Devolviendo la funcion los buffers que han recibido mensaje y el contenido de los mensajes en formato StdFrame.

```
byte CAN::WAITRXN(StdFrame *CAN_RXB0, StdFrame *CAN_RXB1)
{
   byte Status;
   StdFrame FrameBuffer;
   int timeout=1000;

   for(int i=0; i<timeout; i++)         //Wait until timeout and Check Status
   {    Status=STATUSRX();
        if(Status != 0) break;
        delay(1);
    }

    if(Status==0) return Status;
    else if(Status==1) { RXNREAD(0x01,CAN_RXB0);    return Status;}
    else if(Status==2){ RXNREAD(0x02,CAN_RXB1);    return Status;}
    else if(Status==3){ RXNREAD(0x01,CAN_RXB0); RXNREAD(0x02,CAN_RXB1);
                        return Status;}
   }
```

Y el segundo método implementado, es mediante interrupciones. Un método mas eficiente al no tener que supervisar continuamente el estado de los buffers. El microcontrolador recibe las interrupciones RXB0INT y RXB1INT generadas en el MCP2515. Para poder utilizar este método las interrupciones deben estar habilitadas en el registro CANINTE del MCP2515 y generar una interrupción el el programa del microcontrolador.

Las primeras funciones son para habilitar y deshabilitar las interrupciones del microcontrolador relativas a la recepción de mensajes.

```
void INT_CANREAD_ON(void)
{
attachInterrupt(digitalPinToInterrupt(RX0BF), GET_RX0, FALLING);    //pin 2 =RXB0F
attachInterrupt(digitalPinToInterrupt(RX1BF), GET_RX1, FALLING);    //pin 3= RXB1F
```

```
 }
 void INT_CANREAD_OFF(void)
 {    detachInterrupt(digitalPinToInterrupt(RX0BF));
      detachInterrupt(digitalPinToInterrupt(RX1BF));
   }
```

Las funciones asociadas a las interrupciones son GET_RX0 y GET_RX1 que utilizan el comando RXNREAD para descargar el buffer asociado a la interrupción.

```
void GET_RX0(void)
{       CAN.RXNREAD(0x00,&BRX0);
 }


void GET_RX1(void)
{      CAN.RXNREAD(0x01,&BRX1);
   }
```

Este ultimo método es el mas rápido para la recepción de mensajes CAN.
Con esta ultima se completan las funciones dedicadas para la transmisión y recepción de mensajes CAN. Se resumen a continuación.

Resumen de funciones para la transmisión y recepción de mensajes CAN.

```
void CAN::LOADTX(byte TXBN, StdFrame CanMsg)
byte CAN::SENDTXBn(byte TXBN)
byte CAN::WAITRXN(StdFrame *CAN_RXB0, StdFrame *CAN_RXB1)
void INT_CANREAD_ON(void)
void INT_CANREAD_OFF(void)
void GET_RX0(void)
void GET_RX1(void)
```

3.12 Funciones CANopen.

El siguiente listado de funciones son las implementadas para el protocolo CANopen, funciones necesarias para la configuración de los sensores CANopen objetivo final del proyecto y dos funciones adicionales que distinguen este proyecto de su competidor, función de autodetección de baud rate y detección de numero de nodo del sensor .

La primer función diseñada no esta relacionada con el protocolo CANopen pero si con el control del equipo y utilizada durante el proceso de configuración de los sensores CANopen. Mediante el control de un relé se corta la alimentación durante unos segundo a los sensores CANopen, para hacer un reset del equipo.

3.12.1 ONOFF().

Durante la configuración de los sensores CANopen, para que los cambios se hagan efectivos es necesario hacer un reset del equipo CANopen. La función ONOFF() actúa sobre un relé que apaga el dispositivo durante 2 segundos.

```
void CAN::ONOFF(void)
{    pinMode(PINRELE, OUTPUT);            // set pin as output
     digitalWrite(PINRELE, HIGH);         //Turn OFF CAN Device
     delay(2000);                         //Wait two seconds
     digitalWrite(PINRELE,LOW);           //Turn ON    CAN Device
     delay(2000);                         //Wait two seconds
}
```

3.12.2 NMTSERVICE.

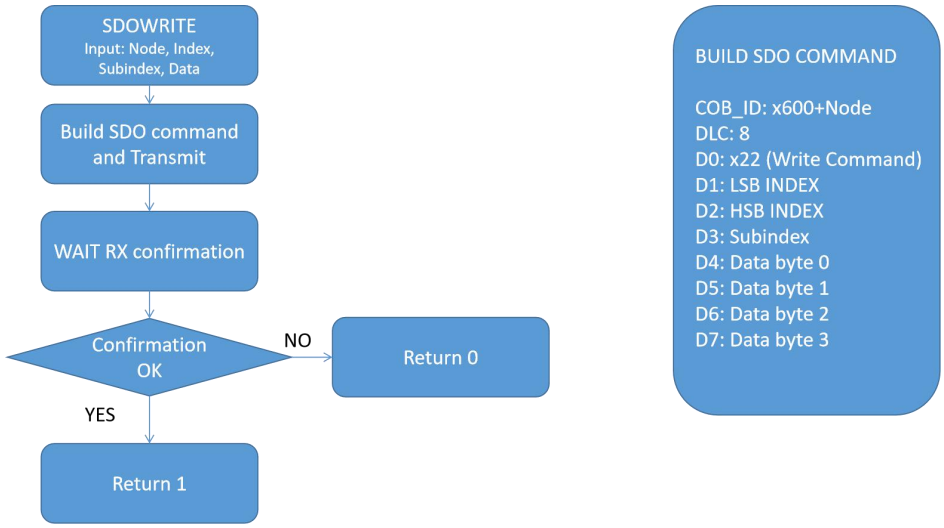
Este comando utiliza el protocolo NMT de CANopen, para cambiar el estado de operación del dispositivo. La función recibe como parámetro un string con el estado de operación deseado (OPERATIONAL", "STOP", "PRE-OPERATIONAL", "RESET-NODE", "RESET-COMMUNICATION"). Y como segundo parámetro el numero de nodo al que va dirigido el comando. Si el el comando NMT va dirigido a todos los nodos el valor de node debe ser 0. La función construye la trama de comunicación CAN para el comando NMT y mediante la función LOADTX y SENDTXBN transmite el comando.

```
void CAN::NMTSERVICE(char NMTC[],byte Node)
{
   StdFrame CANTX;
   CANTX.COB_ID=0;
   CANTX.RTR_BIT=0;
   CANTX.DLC=0x02;
   CANTX.DATA[2]; //={0x00,0x00};

     if (NMTC== "OPERATIONAL")                  CANTX.DATA[0]=0x01;
     else if (NMTC=="STOP")                     CANTX.DATA[0]=0x02;
     else if (NMTC=="PRE-OPERATIONAL")          CANTX.DATA[0]=0x80;
     else if (NMTC=="RESET-NODE")               CANTX.DATA[0]=0x81;
     else if (NMTC=="RESET-COMMUNICATION")      CANTX.DATA[0]=0x82;

     CANTX.DATA[1]=Node;

   LOADTX(TXB_0,CANTX);
   SENDTXBn(TXB_0);

 }
```

3.12.3 SDOWRITE.

Para escribir sobre el diccionario de objetos durante el proceso de configuración de un dispositivo CANopen, se utiliza la función SDOWRITE. La función esta basada en los comandos SDO (service data object) del protocolo CANopen.
La función recibe como parámetros: Numero de nodo, Indice, Subindice y Datos.
En base a los parámetros de entrada se construye el comando SDO, se transmite y se espera confirmación de que el dato ha sido escrito correctamente en el diccionario de objetos. La función puede resumirse en el siguiente diagrama de flujo.

Figura 3-67 Algoritmo comando SDO.



3.12.4 LSS CONFIGURATION.

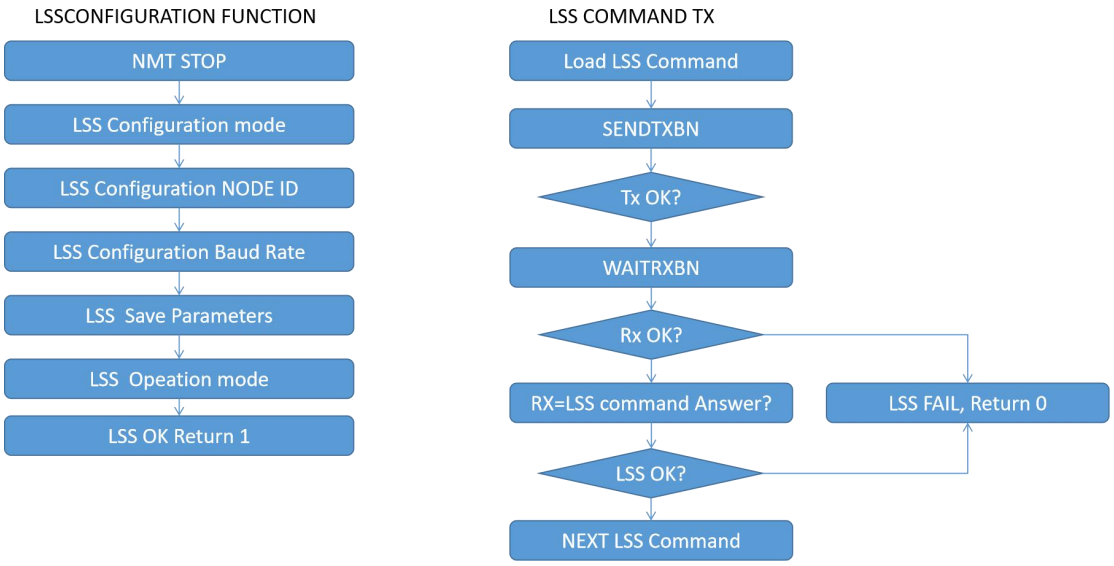
Esta función utiliza el protocolo LSS de CANopen, para configurar el Baud rate y el numero de nodo del dispositivo CANopen. Utiliza la secuencia definida en tabla 3.17, donde se describe el conjunto de instrucciones de LSS. La función recibe como parámetros de entrada el baud rate y el numero de nodo con los que genera la secuencia de comandos LSS.

Tabla 3-17 LSS configuración de baud rate y Numero de nodo.

				DATA								
Tx/Rx	COB-ID	RTR	DLC	B0	B1	B2	B3	B4	B5	B6	B7	DESCRIPCION
Tx	0	0	2	x2	0							NMT Command : STOP
Tx	x7E5	0	8	x4	1	0	0	0	0	0	0	LSS Configuration Mode
Tx	x7E5	0	8	x11	Node-ID	0	0	0	0	0	0	LSS Config Node-ID
Rx	x7E4	0	8	x11	0	0	0	0	0	0	0	LSS Node-ID confirmation
Tx	x7E5	0	8	x13	0	Baud Rate	0	0	0	0	0	LSS Set Baud rate
Rx	x7E4	0	8	x13	0	0	0	0	0	0	0	LSS Baud rate confirmation
Tx	x7E5	0	8	x17	0	0	0	0	0	0	0	LSS Save parameters
Rx	x7E4	0	8	x17	0	0	0	0	0	0	0	LSS Save confirmation
Tx	x7E5	0	8	x4	0	0	0	0	0	0	0	LSS Operation mode

Como se muestra en la Fig. 3-68, la función transmite secuencialmente cada comando, siguiendo para cada comando la secuencia de transmisión de comando LSS. En l que se comprueba que el comando haya sido ejecutado.

Figura 3-68 Secuencia de la función LSSCONFIGURATION.



Las siguientes dos funciones AUTOBAUDRATE y NODE_DETECT, son las que dan a este equipo una de las ventajas sobre el equipo utilizado actualmente. Estas funciones son importantes en el proceso de configuración. Tanto el Dispositivo como el equipo de configur-ación deben transmitir con el mismo baud rate para entablar comunicación y el numero de nodo del dispositivo debe conocerse para poder dirigir correctamente los comandos SDO. Cada fabricante decide el numero de nodo y baud rate por defecto, la detección automática de esta configuración facilita la configuración de los sensores.

Al ser funciones mas complejas, como en la función anterior, para la descripción de su funcionamiento se utilizan diagramas de flujo en vez de código.

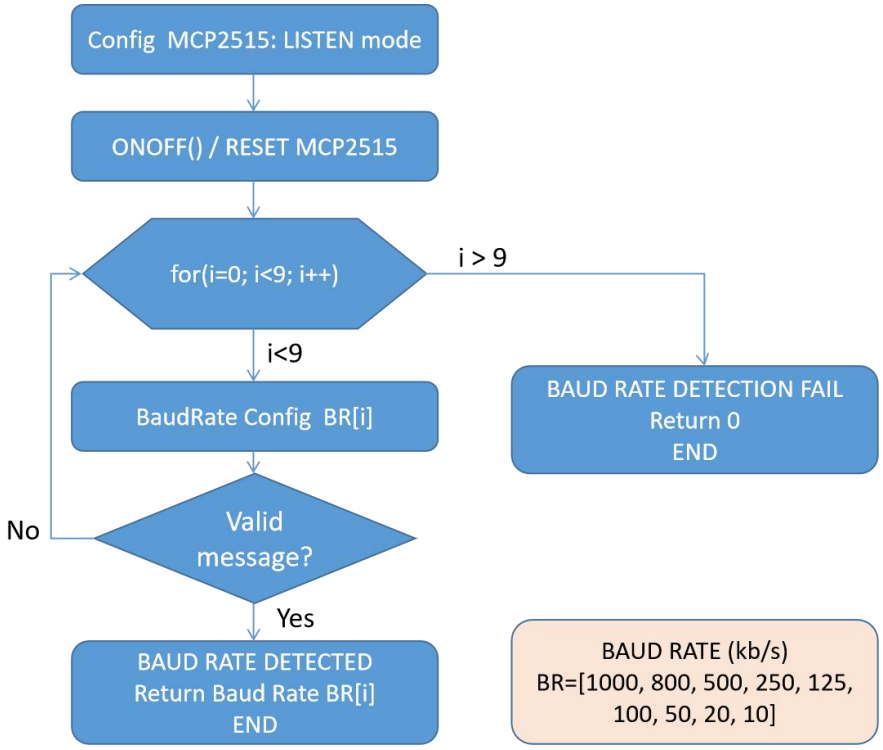
3.12.5 AUTOBAUDRATE.

Esta función detecta automáticamente y de forma muy rápida el baud rate del dispositivo al que este conectado.

El algoritmo de la Fig. 3.68, configura el MCP2515 en modo LISTEN(Escucha). En este modo el controlador recibe cualquier tipo de mensaje el los buffers de recepción RXB0 y RXB1, aun aquellos con errores. En modo LISTEN si un mensaje es recibido no modifica el bit ACK de la trama de comunicación CAN, que es el bit que informa al emisor que el mensaje ha sido recibido.
La siguiente operación del algoritmo apagar y encender el Dispositivo CAN, quitando la alimentación temporalmente con la función ONOFF(). Cuando el sensor enciende envía un mensaje de emergencia (COB-ID: 80+Node), notificando a la red de su presencia. Al no estar preparado para recibir mensajes el controlador, el

dispositivo CAN no encuentra receptor (bit ACK) y seguirá retransmitiendo la trama de comunicación indefinidamente.
El algoritmo entra en un ciclo donde configura el MCP2515 con los valores de baud rate posibles definidos en un array BR[]. El ciclo consiste en configurar el controlador CAN con el primer valor de baud rate, esperar un tiempo y verificar si algún mensaje ha sido recibido en alguno de los buffer RXB0 o RXB1, si algun mensaje ha sido recibido verifica el estado del bit CANINTF.MERRF para comprobar si el mensaje es valido. Si no hay mensajes o el mensaje recibido no es valido, el ciclo se repite con un nuevo valor de Baud Rate. Si en alguno de los ciclos se detecta un mensaje valido, significa que el Baud rate ha sido encontrado y termina la función devolviendo el valor del baud rate, dejando la configuración activa y en modo de operación. Si por el contrario despues de completado el ciclo no se encuentra un valor valido. La función devuelve un 0, indicando que el baud rate no ha sido encontrado.

Figura 3-69 Algoritmo de autodetección de baud rate.



3.12.6 DETECT_NODE.

Esta función detecta el numero de nodo del dispositivo conectado. El baud rate debe estar previamente configurado, de forma manual o con la función autobaudrate. El conocer el numero de nodo, es importante para dirigir correctamente los comandos SDO cuyo COB-ID es : x600 + Nodo. El algoritmo, pone en modo operativo al dispositivo mediante un comando NMT y envía una señal de sincronia COB-ID: x80, el dispositivo responderá con un TPDO (Transmit process data object), que puede ser cualquiera de los mostrados en la siguiente tabla.

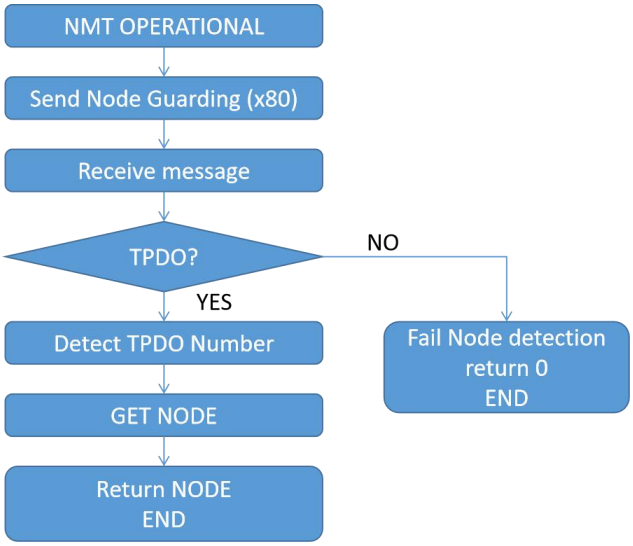
Tabla 3-18 Listado de TPDOS y sus posibles valores de COB-IDs.

TPDOs	COB-ID
TPDO1	x181-x1FF (x180 + Nodo)
TPDO2	x281-x2FF (x280 + Nodo)
TPDO3	x381-x3FF (x380 + Nodo)
TPDO4	x481-x4FF (x480 + Nodo)

Recibido el TPDO, el algoritmo detecta con el COB-ID a que numero de TPDO pertenece y determina el numero de nodo del dispositivo con una simple ecuación.

$$Nodo = COBID - xn80$$

Figura 3-70 Algoritmo de detección de numero de nodo.



Resumen de Funciones para CANopen

```
void CAN::ONOFF(void)
void CAN::NMTSERVICE(char NMTC[],byte Node)
byte SDOWRITE(byte Node,int Index,int SubIndex, long DATA)
long SDOREAD(byte Node,int Index,byte SubIndex)
byte LSSCONFIGURATION(int Node,int BR)
int CAN::AUTO_BAUD_DETECT(void)
int CAN::NODE_DETECT(void)
```

Con estas ultimas quedan cubiertas las funciones principales de la librería para el control del MCP2515 y las del protocolo CANopen

3.13 Control de la memoria SD.

Se utiliza un memoria externa SD donde se guardan los archivos de configuración para cada sensor. Los archivos son de texto con el siguiente formato:

Tabla 3-19 Formato de archivo de configuración.

Formato	Ejemplo
Nombre del dispositivo	MTS PALA 1
Baut rate	250
Numero de nodo	17
Indice Subindice Valor del registro	1800 01 0191
Indice Subindice Valor del registro	1800 02 01
. . .	. . .
Indice Subindice Valor del registro	1010 01 65766173

La primer linea, es solo para identificación del archivo de configuración.
La segunda y tercer linea es baud rate y numero de nodo, utilizados para la configuración mediante el protocolo LSS (layer setting service). Y las siguientes lineas para configuración mediante el protocolo SDO (service data object) para configurar los registros identificados por el Indice y Subindice con el valor del registro, separados todos por un espacio.

Para el control de la memoria SD se utiliza una librería publica en el IDE de arduino. De las disponibles, la utilizada es la libreria SD.h. Que permite hacer operaciones de lectura y escritura sobre la memoria SD. La comunicación entre el microcontrolador y la memoria SD es a través de SPI, por lo que la librería SPI.h debe estar incluida igualmente.

3.13.1 Funciones de la memoria SD.

Inicializacion

La memoria SD debe ser previamente inicializada antes de intentar leer o escribir el comando de la la libreria SD.h tiene l siguiente formato.

```
SD.begin(CS_SDCARD)
```

Abrir archivo

El siguiente comando para abrir un archivo de texto desde la memoria SD es el siguiente:

```
myFile = SD.open(FileName, FILE_READ)
```

Donde myFile es una variable de tipo File declarada previamente, FileName es un una variable del tipo char array, y FILE_READ el comando de la librería SD para abrir el archivo en modo lectura.

3.14 RTC (real Time Clcok).

En esta etapa del proyecto no se utiliza y el integrado esta montado para futuras aplicaciones de datalogger donde tener una medición de tiempo real resulta útil.
La librería a utilizar como referencia seria RTClib.h, que requiere de la librería Wire.h, necesaria para la comunicación I2C.

3.15 FTDI232.

Para el caso del FTD232 no se requiere librerías, esta conectado al puerto serie a través del cual se programa el microcontrolador ATMEL2560, lo necesario para descargar el programa esta contenido en el bootloader de Arduino con el que esta programado. Para enviar y recibir mensajes a través del puerto USB se utilizan los comandos :

Para inicializar el puerto:

```
Serial.begin(baud rate);
```

Para escribir en el puerto (Función útil para la depuración del programa):

```
Serial.println("Texto");    Envía texto con salto de linea al puerto Serie (USB)
Serial.print("Texto");      Envía texto sin salto de linea al puerto Serie(USB)
```

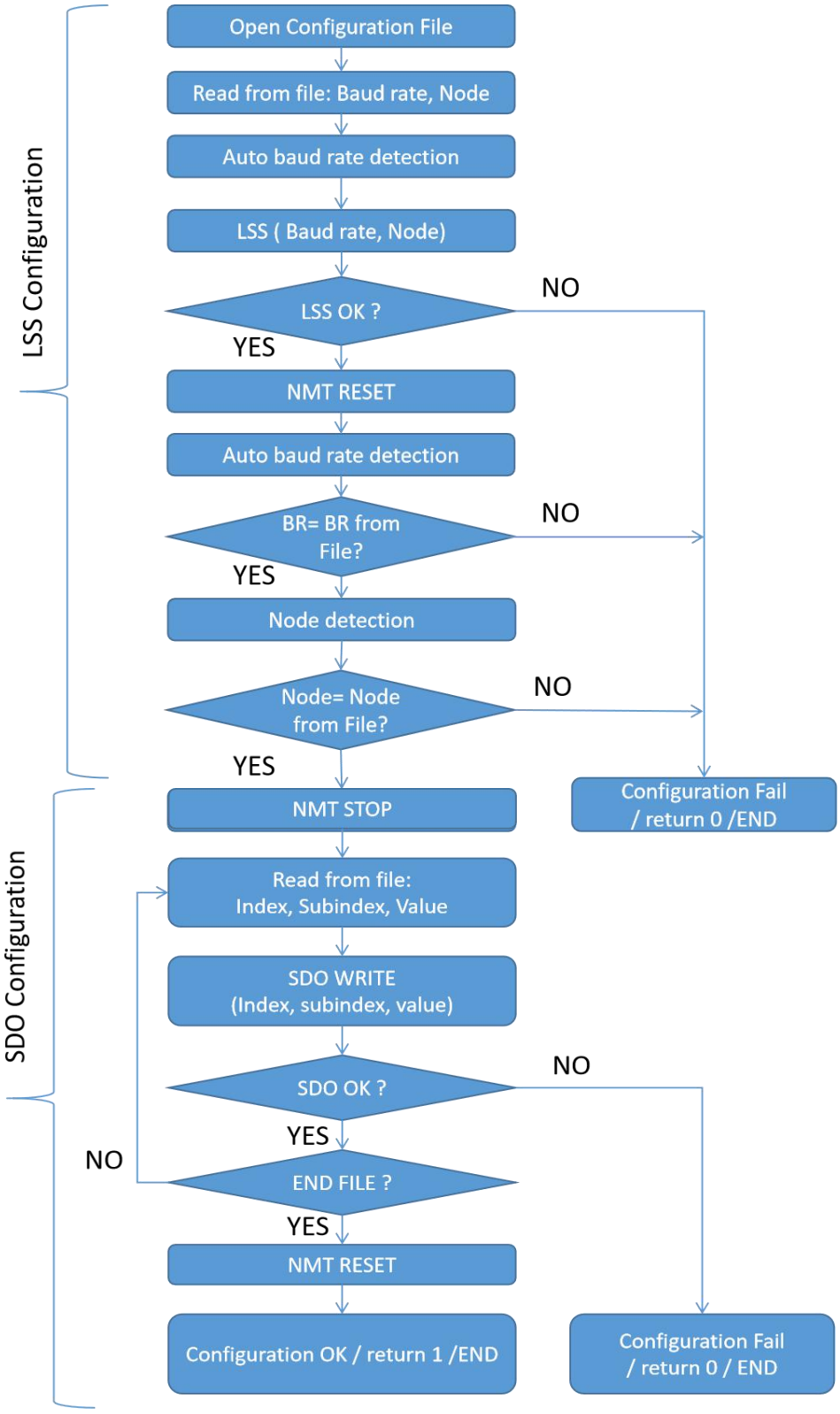
3.16 Algoritmo de configuración de sensores CANopen.

Completadas las funciones para el control de los dispositivos del equipo, las diversas funciones y librerías serán utilizadas para generar el algoritmo para la configuración de los sensores CANopen.
Con el equipo en operación, una vez seleccionado el dispositivo CANopen desde la función Menú del equipo, el dispositivo tiene un nombre de archivo asociado guardado en la memoria SD, que lo enviara como parámetro a la función Config_Sensor, diseñada para leer desde el archivo, con el formato antes descrito, los parámetros con el que el dispositivo sera configurado.

El algoritmo de configuración, abre el archivo de texto y lee linea por linea su contenido mientras ejecuta comandos CANopen configurando el dispositivo.
El programa inicia extrayendo las dos primeras lineas del archivo, Baud rate y Node. Detecta el baud rate actual del dispositivo, para posteriormente mediante comandos LSS configurar el equipo con e nuevo Baud rate y numero de nodo.
Mediante un comando NMT hace un reset del dispositivo para cargar la nueva configuración y comprueba que el Baud rate y el numero de nodo, sean los configurados.
Las siguientes lineas del archivo de configuración, son las actualizaciones del diccionario del dispositivo, por linea separados por un espacio se encuentra el Indice,

subindice y el valor. Utilizando el comando SDOWRITE, configura el diccionario de objetos.

Figura 3-71 Algoritmo configuración de dispositivos CANopen.



4. Resultados.

Se construyeron 2 prototipos para comprobar el funcionamiento del Hardware y del programa desarrollado. El montaje de los componentes de SMT y TH fueron montados manualmente.

El mecanizado para solo dos cajas de aluminio resultaba demasiado caro, para la construcción de los prototipo, por lo que se remplazaron las caras a mecanizar por piezas de PLA hecha en una impresora 3D. Lo que resulta de extrema utilidad para la fabricación de prototipos, en tiempo y costo, en los que se pueden comprobar el diseño mecánico de las piezas previas al mecanizado final. El diseño mecánico para el primer prototipo resulta con medidas exactas, impresas y montadas las piezas, por las pequeñas diferencias que existen entre el diseño mecánico 3D y las piezas reales, se observa que se requiere modificar ligeramente las aperturas mas con fines estéticos que funcionales.

Se hicieron pequeños programas específicos para comprobar cada uno de los periféricos del equipo por separado. La electrónica funciono correctamente y no requirió ningún retrabajo o modificación de diseño.

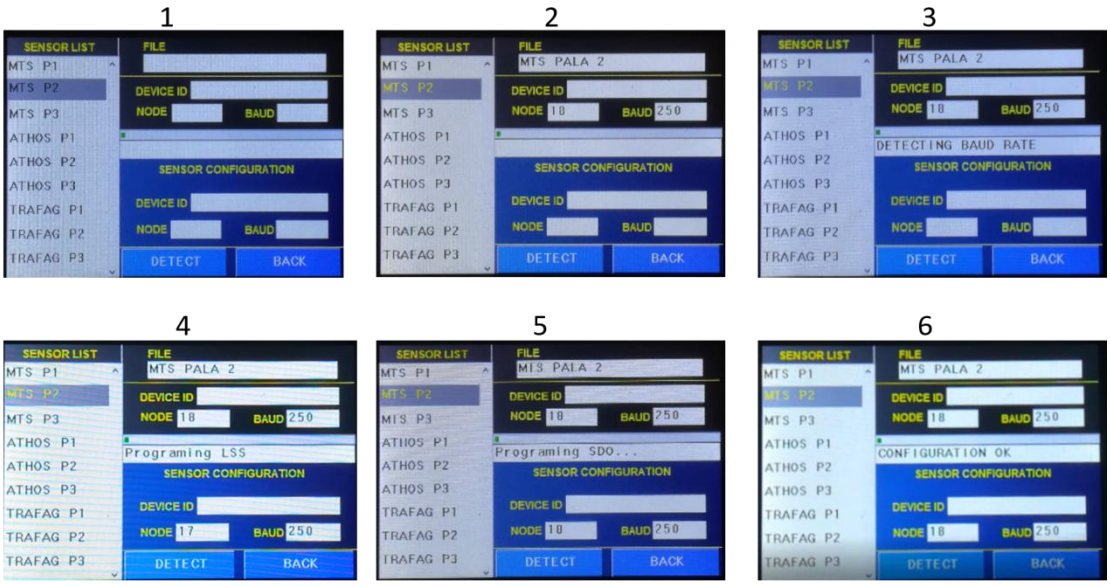
Con los prototipos montados, el programa para la configuración de dispositivos CANopen se fue construyendo y depurando hasta hacerlo funcionar.

4.1 Pruebas de configuración.

El único dispositivo CANopen que se tiene disponible para las pruebas de configuración es un sensor MTS, la conexión con el equipo de configuración se muestra en la Figura 4-1. El proceso de configuración puede verse en la secuencia de imágenes de la pantalla del equipo, Figura 4-2.

Figura 4-1 Equipo de configuración y sensor MTS.

Figura 4-2 Interfaz gráfica del equipo durante la configuración



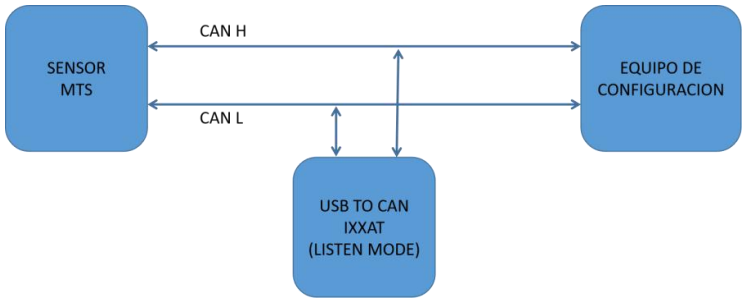
Cuya secuencia se explica a continuación.:

Proceso de configuración Sensor MTS P2.

1. Utilizando el mando U/Down se navega hasta el sensor MTS P2.
2. Una vez que se esta en la posición el primer ENTER carga los datos del archivo en la sección superior derecha.
3. Con el Segundo ENTER se inicia la configuración del sensor MTS con los parámetros de la pala 2. En donde la primer tarea es detectar automáticamente el baud rate y el numero de nodo.
4. Baud rate y numero de nodo han sido detectados (250kb/s, nodo 17) e inicia la nueva configuración con el protocolo LSS. Donde se vuelve a la rutina de autodetección para comprobar la configuración.
5. Ahora el numero de nodo es 18 y se inicia con la configuración de los registros del diccionario con el protocolo CANopen.
6. Una vez completada la modificación de todos los objetos del diccionario, se salvan los cambios y se da por terminada la configuración.

Para comprobar el proceso de configuración se ha utilizado un convertidor de USB a CAN y el software PCAN, en modo de escucha con el que se registra el trafico entre el Sensor MTS y el equipo.

Figura 4-3 Conexión IXXAT para registro de trafico en el CAN bus.



Los resultados de las pruebas se muestran en la tabla 4.1, donde el trafico de la red CAN es analizado en la columna de la derecha.

Tabla 4-1 Registro Configuración CANopen.

No. MSG	Time ms	Rx/Tx	COB-ID(HEX)	DLC	Datos (HEX)								Analisis
5543	5,0995	Rx	93	0									Comando ON/OFF
5544	5,0998	Rx	93	0									Autobaudrate
5545	5,1	Rx	93	0									
5543	5,0995	Rx	93	0									
5544	5,0998	Rx	93	0									
5545	5,1	Rx	93	0									Baud rate configurado.
5546	5,201	Tx	0	2	1	0							Comando NMT : Operational Mode
5547	5,201	Rx	193	7	F7	20	1	0	FF	FF	0		Deteccion de nodo :
5548	5,251	Tx	80	0									Envio Syncron x80
5549	5,251	Rx	193	7	F8	20	1	0	FF	FF	0		TPDO x193 Nodo=x193-x180=x13(19 MTS Pala 3)
5550	5,289	Tx	0	2	2	0							NMT STOP
5551	5,489	Tx	0	2	2	0							Inicia configuracion LSS
5552	5,589	Tx	7E5	8	4	1	0	0	0	0	0	0	LSS modo de configuracion
5553	5,689	Tx	7E5	8	11	12	0	0	0	0	0	0	LSS configura nodo x12(18 MTS pala 2)
5554	5,69	Tx	7E4	8	11	0	0	0	0	0	0	0	LSS confirmacion desde el sensor MTS
5555	5,791	Tx	7E5	8	13	0	3	0	0	0	0	0	LSS configura Baud rate x3 =250kb/s
5556	5,791	Rx	7E4	8	13	0	0	0	0	0	0	0	LSS MTS confirma cambio de baud rate
5557	5,892	Tx	7E5	8	17	0	0	0	0	0	0	0	LSS Salvar parametros
5558	5,958	Rx	7E4	8	17	0	0	0	0	0	0	0	LSS MTS parametros salvados
5559	5,993	Tx	7E5	8	4	0	0	0	0	0	0	0	LSS modo operativo
5560	6,193	Tx	0	2	81	0							NMT Reset Node
5561	6,21	Rx	92	0									MTS envia estado COBID x92
5562	6,211	Rx	92	8	0	0	0	0	0	0	0	0	
5563	6,599	Tx	612	8	22	0	18	1	92	1	0	0	Inicia configuración SDO:
5564	6,6	Rx	592	8	60	0	18	1	0	0	0	0	SDO Configuración paramettros del TPDO 1 Index 1800
5565	7,403	Tx	612	8	22	0	18	2	1	0	0	0	
5566	7,404	Rx	592	8	60	0	18	2	0	0	0	0	
5567	8,207	Tx	612	8	22	0	1A	0	0	0	0	0	SDO Configuracion Mapping TPDO1 Index: 1A00
5568	8,207	Rx	592	8	60	0	1A	0	0	0	0	0	
5569	9,011	Tx	612	8	22	0	1A	1	20	1	20	60	
5570	9,011	Rx	592	8	60	0	1A	1	0	0	0	0	
5571	9,815	Tx	612	8	22	0	1A	2	10	1	30	60	
5572	9,816	Rx	592	8	60	0	1A	2	0	0	0	0	
5573	10,619	Tx	612	8	22	0	1A	3	8	1	0	63	
5574	10,62	Rx	592	8	60	0	1A	3	0	0	0	0	
5575	11,423	Tx	612	8	22	0	1A	0	3	0	0	0	
5576	11,424	Rx	592	8	60	0	1A	0	0	0	0	0	
5577	12,227	Tx	612	8	22	0C	10	0	F4	1	0	0	SDO Configuracion Guard time Index: 100C
5578	12,227	Rx	592	8	60	0C	10	0	0	0	0	0	
5579	13,031	Tx	612	8	22	0D	10	0	3	0	0	0	SDO Configuracion Life time factor Index:100D
5580	13,031	Rx	592	8	60	0D	10	0	0	0	0	0	
5581	14,833	Tx	612	8	22	10	10	1	73	61	76	65	SDO Salvar configuración
5582	15,197	Rx	592	8	60	10	10	1	0	0	0	0	MTS confirma configuracion salvada.
5583	15,334	Tx	0	2	81	0							NMT Reset Node Fin de la configuración
5584	15,351	Rx	92	0									
5585	15,352	Rx	92	8	0	0	0	0	0	0	0	0	

La captura del trafico de la red CAN, coincide con la secuencia esperada para la configuración de los sensores MTS. Donde la comunicación y configuración del dispositivo CANopen se completa correctamente.

El tiempo de configuración es de 15 segundos aproximadamente, comparado con el equipo que se utiliza actualmente es extremadamente rápido, en donde el tiempo de configuración puede llegar a mas de 5 minutos, con intervenciones del técnico en varias ocasiones, sin considerar el tiempo de arranque de Windows y de acceso al programa.

5. Conclusiones.

CANopen es un protocolo de comunicación muy robusto a nivel de señal y a nivel de protocolo donde existen varios mecanismos de supervisión de errores que asegura la integridad de los datos transmitidos. Su robustez y sus características técnicas las hace apropiadas para aplicación en aerogeneradores.
Los sensores CANopen son de mayor complejidad que los sensores analógicos y requieren ser configurados previos a su instalación, para ello se utiliza un equipo de configuración de sensores CANopen. En el ámbito del sector eólico, la portabilidad es muy importante para el técnico de mantenimiento. Con el equipo diseñado en este proyecto se logran los objetivos de obtener un equipo portátil que no requiera de ordenador, que sea fácil de utilizar y que efectúe de forma rápida y confiable la configuración de dispositivos CANopen.

5.1 Trabajo Futuro.

El equipo presentado es un primer prototipo que ha demostrado su funcionalidad, pero que debe ser mejorado en algunos aspectos.
En cuanto a Hardware, hay algunos puntos a menorar, durante las pruebas, en un par de ocasiones se ha llegado a dañar el controlador CAN, pensando en el entorno de trabajo, donde hay bastante EMIs, es recomendable cambiar el transductor CAN MCP2551, por uno con aislamiento galvánico que proteja la electrónica interna del equipo, un integrado posible puede ser el MAX14882.
La memoria SD sobresale demasiado, para un segundo prototipo es recomendable desplazar 2 mms hacia el interior.

Sobre el programa del microcontrolador, durante la descripción de las librerías diseñadas para el proyecto, ha permitido detectar varios puntos en los que el algoritmo puede ser optimizado. Existen aun algunos Bugs que deben ser eliminados, aun así en términos generales el equipo funciona correctamente

El objetivo final del proyecto es que este equipo no solo sea un equipo de configuración de sensores CANopen. Si no que sirva a su vez como equipo para el diagnostico de la red CAN bus de lo aerogeneradores de Acciona Winpower AW3000. Las herramientas futuras a implementar en el mismo equipo, una prueba de calidad de linea y un datalogger que registre mensajes de emergencia y problemas de node guarding que ayuden a diagnosticar problemas de la red CAN, que a la fecha son invisibles para el técnico.

Como segundo objetivo, es poder hacer mas genérico al equipo de configuración, compatible con archivos DCF (Device configuration File) para que pueda ser utilizado en la industria para configurar cualquier dispositivo CANopen.

Bibliografía.

[1] Pfeiffer Olaf,Ayre Andrew,Keydel Christian(2008). *Embedded Networking with CAN and CANopen. Editorial: Copperhill Technologies Corporation. Greenfield USA.*

[2] CAN in Automation (CIA) (2011), *CIA301 CANopen application layer and Communication profile Ver. 4.2.0.*

[3] CAN in Automation(CIA)(2008), CIA401 Device profile for generic I/O modules Ver. 3.0.0

[4] CAN in Automation(CIA) (2008), *CIA305 Layer setting service Ver. 1*

[5] Corrigan Steve(2016). *Introduction to the Controller Area Network (CAN).* Texas instruments Application report.

[6] Watterson Conal. *Controller area Newtork(CAN) Implementation Guide.* Analog Devices Application note AN-1123.

[7] Esd electronic systems design gmbh(2007). *CAN-Bus Troubleshooting guide.* ESD electronic INC.

[8] Richards Pat(2002). *A CAN physical layer discussion.* Microchip application note AN228.

[9] *Programming instructions for digital electrohydraulics with CAN bus interface.* ATOS User manual.

[10] Temposonic R-Series. *MTS CAN bus interface CANopen Data protocol 304*. MTS Sensor Technologies.

[11] *Trafag operation instruction software Ver 1.2.* Trafag AG.

[12] *MCP2515 Stand Alone CAN controller with SPI interface(Datasheet) (2012).* Microchip Technologie INC.

[13] *MCP2551 High Speed Transceiver.(2010)* Microchip Technologies INC.

[14] Sánchez Díaz Raquel. *CANopen.* Departamento de informática y automática. Facultad de ciencias Universidad de Salamanca.

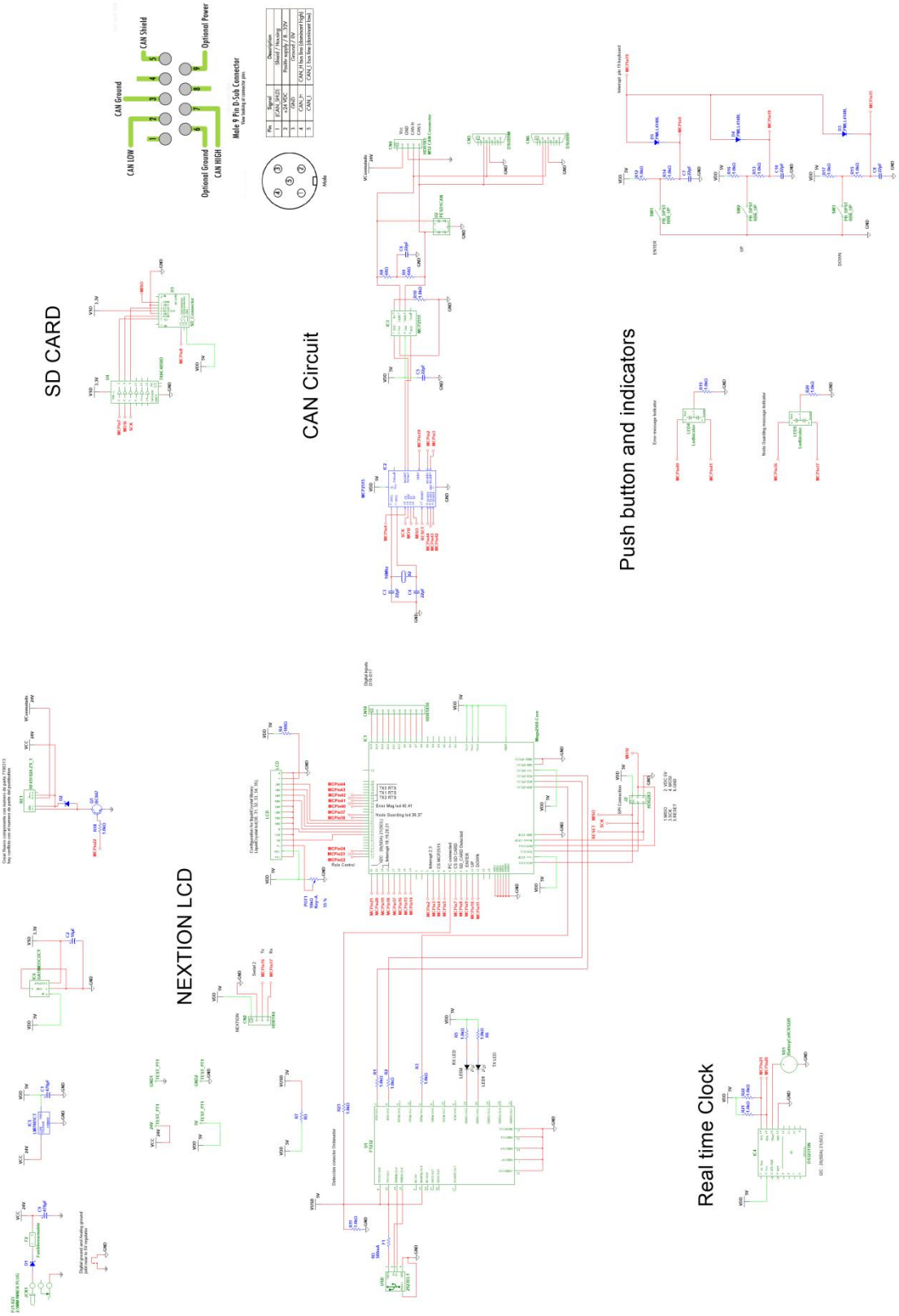
[15] *Herramienta de configuración nodos CAN.* Manual aerogeneradores AW3000, Doc. IU0189.

[16] https://www.kvaser.com/can-protocol-tutorial/

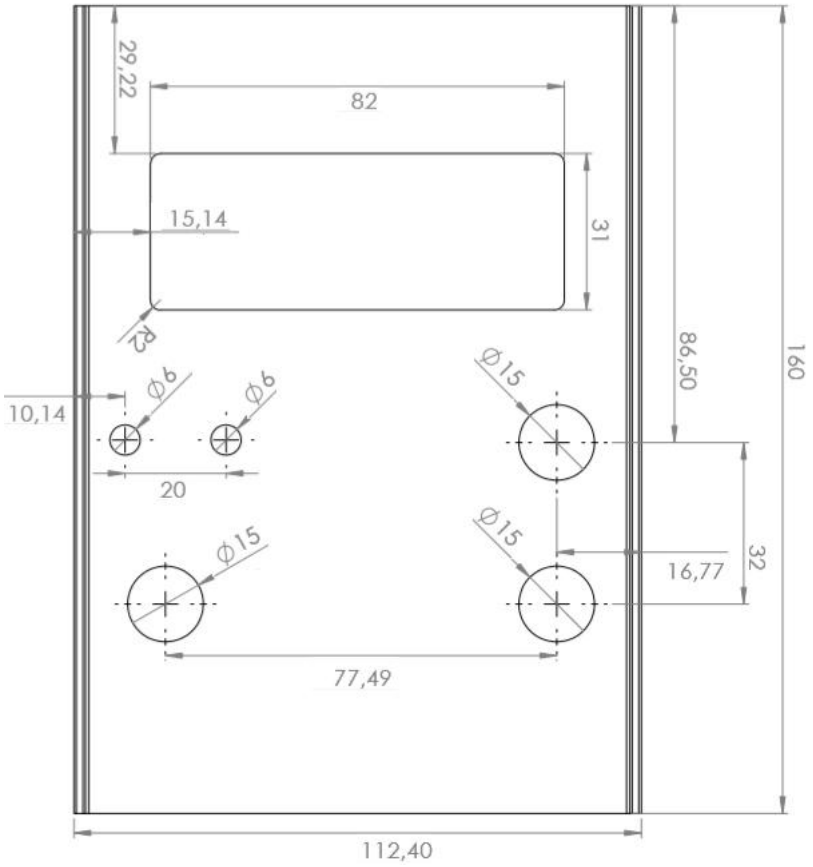
[17] https://www.kvaser.com/about-can/higher-layer-protocols/canopen/

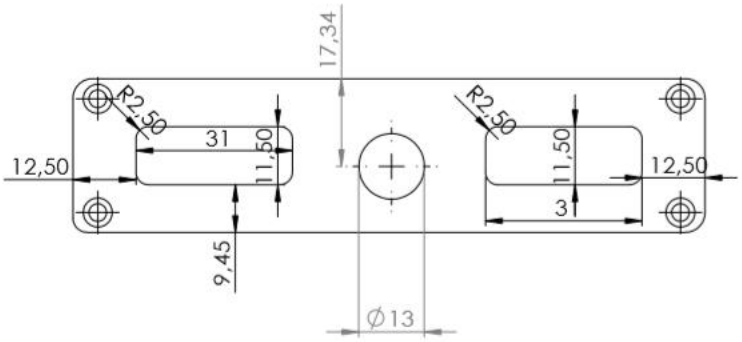
[18] https://www.can-cia.org/
[19] Instituto Schneider electric de formación(2008). *Manual CANopen.* Schneider Electric. Schneider electric España.

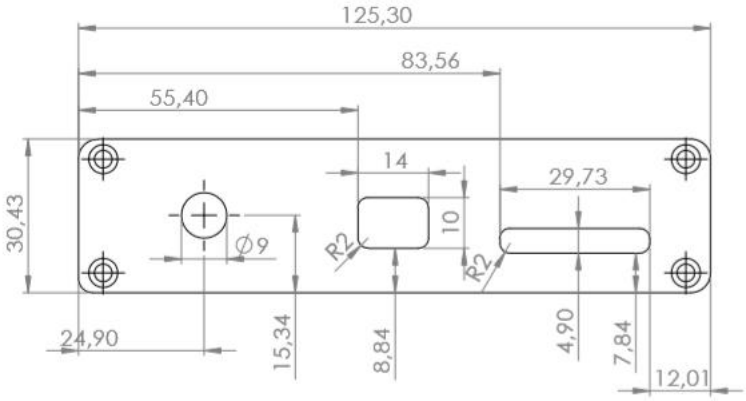
APENDICE A. Diseño eléctrico equipo de configuración CAN.



APENDICE B. Mecanizado.







APENDICE C. Bill de materiales.

RefDes	Vendor	Vendor Part No.	Unit price	Uds	SubTotal
IC2	RSonline	628-3548	1,58	1	1,58
IC3	RSOnline	738-6036	0,97	1	0,97
USB	RSonline	748-0866	1,42	1	1,42
C3, C4, C5, C6, C7, C8, C10	RSOnline	669-8482	0,093	7	0,651
X2	RSonline	852-4872	0,35	1	0,35
CN4	RSonline	914-4493	12,29	1	12,29
U1	RSOnline	888-8732	3,79	1	3,79
R1, R2, R3, R5, R6, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23	RSonline	223-2265	0,17	19	3,23
J2					
RE1	RSOnline	719-6313	1,16	1	1,16
U2	RSonline.es	508-860	0,286	1	0,286
R8, R9	RSOnline	812-7119	0,015	2	0,03
IC1	RSonline	769-7418	59	1	59
LCD	RSOnline	532-6818	30	1	30
POT1	RSOnline	0473473	0,402	1	0,402
IC5	RSonline	666-4379	5,5	1	5,5
C9	Rsonline	862-3250	0,15	1	0,15
F2	Rsonline	517-6809	0,42	1	0,42
IC6	RSonline	661-6733	0,378	1	0,378
U3	RSOnline	685-0779	3,74	1	3,74
U4	RSOnline	483-6544	0,156	1	0,156
C1	RSOnline	855-6129	0,16	1	0,16
C2	RSOnline	855-6129	0,16	1	0,16
SW1, SW2, SW3	RSOnline	719-6313	1,16	1	1,16
LED4, LED5	RSOnline	228-5764	0,23	1	0,23
IC4	RSOnline	732-7582	8,59	1	8,59
SB1	RSOnline	430-653	2,37	1	2,37
CN6	RSOnline	495-9760	1,252	1	1,252
CN5	RSOnline	495-9805	0,8	1	0,8
JCK1	RSOnline	122-4883	0,9	1	0,9
F1	RSOnline	787-4209	0,297	1	0,297
LED1, LED2	RSOnline	466-3908	0,092	2	0,184
R4	RSOnline	223-2120	0,009	1	0,009
D1, D2	RSOnline	792-5640	0,58	2	1,16
Q1			0,25	1	0,25
CN10					0
5V, 24V, GND1, GND2	RSonline	434-784	0,982	4	3,928
					0
Alluminium box	RSonline	8180570	23,59	1	23,59
					0
Mecanizado			25	1	25
				TOTAL:	**195,543**

APENDICE D. Código de programa.

Librería CANMEGA.h

```
/*CANL Library Edgar Bueno Hernandez. 10/02/2018    */

#ifndef CANMEGA_h
#define CANMEGA_h
#include <Arduino.h>
#include "CANMEGAGlobalVars.h"

class CAN
{     public:
            CAN();      //Constructor
     void CANMEGA_CONFIG(void);
     void RESET(void);
     byte READ(byte Address);
     void READ_M(byte Address, byte *AData, byte Ndatos);
     void READ_RX_BUFFER(byte Address, byte *RxData, byte Ndata);
     void RXNREAD(byte RXbuffer, StdFrame *Can_Msg);
     void WRITE(byte Address, byte Data);
     void WRITE_M(byte Address, byte DataArray[], byte Ndata);
     void LOADTX(byte TXBN,    StdFrame CanMsg);
     void RTS(byte TXBN);
     byte READ_STATUS(void);
     byte RX_STATUS(void);
     void BIT_MODIFY(byte Address, byte Mask, byte Data);
     void SETMODE(byte MODE,byte OSM);
     String GETMODE(void);
     void SPEEDCONFIG(int BAUDRATE);
     void STATUSRXDATA(byte *RxBuffer, byte *MsgType, byte *FilterMatch);
     void CLEARBUFFERRXN(byte RXN);
     void RXBNCTRL_CONFIG(byte RX0_MODE, byte RX1_MODE, byte ROLLOVER);
     void SETMASK_RXMN(byte RXMASKN_SIDH, unsigned int MASK_HEX);
     void SETFILTER_RXFN(byte RXFILTERN_SIDH, unsigned int FILTER_HEX);
     byte SENDTXBn(byte TXBN, StdFrame CanMsg);
     byte NMTSERVICE(byte NMTC,byte NODE);
     void ONOFF(void);
      int AUTO_BAUD_DETECT(void);
      void CANBAUDCONFIG(int Baud);
      byte WAITRXN(StdFrame *CAN_RXB0,StdFrame *CAN_RXB1);
      byte STATUSRX (void);
      int NODE_DETECT(void);

     private:
     };
```

#endif

CANMEGA.cpp

```
/*
  Morse.cpp - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/
#include <Arduino.h>#include <SPI.h>
#include "CANL.h"#include "mcp2515_defsL.h"#include "CANGlobalVarsL.h"

void CAN::SPI_CONFIG(byte CS)
{
  pinMode (CS, OUTPUT);
  // initialize SPI:  SPI.beginTransaction(SPISettings(14000000, MSBFIRST,
SPI_MODE0));
  SPI.begin();
}

void CAN::RESET(void)
{
  digitalWrite(CS, LOW);
  SPI.transfer(SPI_RESET);
  digitalWrite(CS, HIGH);
  delay(5); //Let MCP2515 return to the normal operation after RESET}

//-------------------------------------------------------------
//Simple function Read Data over the address return Data readed
byte CAN::READ(byte Address)
{
  byte Data;
  digitalWrite(CS, LOW);

  SPI.transfer(SPI_READ);
  SPI.transfer(Address);

  Data = SPI.transfer(0x00);

  digitalWrite(CS, HIGH);
  return Data;
}
//-----------------------------------------------------------------//This is the same
function than CAN_READ but it is able to read n Data after Address pointer
void CAN::READ_M(byte Address, byte *AData, byte Ndatos)
{
  digitalWrite(CS, LOW);

  SPI.transfer(SPI_READ);
  SPI.transfer(Address);
  for (byte i = 0; i < Ndatos; i++)
  {
    AData[i] = SPI.transfer(0x00);
  }
```

```
  digitalWrite(CS, HIGH);}
//----------------------------------------/* CAN_Byte Rx Buffer
Instruccion 10010nm0
n m                Address point     Address
0 0   0   0x0        RXB0SIDH      0x61
0 1   0   0x2        RXB0D0        0x66
1 0   0   0x4        RXB1SIDH      0x71
1 1   0   0x6        RXB1D0        0x76

Address Could Be  RXB0SIDH(0x61), RXB0D0(0x66), RXB1SIDH(0x71),RXB1D0(0x76)
the CAN_READ with address RXB0SIDH.. RXB1D0 function can replace CAN_READ_RX_Buffer but
with this function the message its shorter

*/
void CAN::READ_RX_BUFFER(byte Address, byte *RxData, byte Ndata)
{
  byte ReadRxBuffer;

  switch (Address) {
    case RXB0SIDH:   ReadRxBuffer = SPI_READ_RX | 0x0; break;
    case RXB0D0:     ReadRxBuffer = SPI_READ_RX | 0x2; break;
    case RXB1SIDH:   ReadRxBuffer = SPI_READ_RX | 0x4; break;
    case RXB1D0:     ReadRxBuffer = SPI_READ_RX | 0x6; break;
    default:         ReadRxBuffer = SPI_READ_RX | 0x0; break;
  }

  digitalWrite(CS, LOW);

  SPI.transfer(ReadRxBuffer);
  for (byte i = 0; i < Ndata; i++)
  {
    RxData[i] = SPI.transfer(0x00); //to generate clock signal to receive data over
MISO  }

  digitalWrite(CS, HIGH);

}

//FAST RX buffer  command  0x00 for RXB0, 0x04 for RXB1void
CAN::FAST_READ_RX_BUFFER(byte command, byte *RxData)
{
         byte DLCN=8;
         digitalWrite(CS, LOW);

         SPI.transfer(SPI_READ_RX |command);                          //ReadRxBuffer);
         for (byte i = 0; i < 13; i++)
         {
                  if(DLCN==0) RxData[i]=0x00;
                  else { RxData[i] = SPI.transfer(0x00); //to generate clock signal
to receive data over MISO            if(i==4) DLCN=RxData[i];
                  if(i>4) DLCN--;}
                  //RxData[i] = SPI.transfer(0x00);               //to generate clock
signal to receive data over MISO     }
```

```
        digitalWrite(CS, HIGH);
}

/*
//FAST RX buffer  command  0x00 for RXB0, 0x04 for RXB1
void CAN::FAST_READ_RX_BUFFER(byte RXBNCTRL, byte *RxData)
{
     byte DLCN=8;

        digitalWrite(CS, LOW);

        SPI.transfer(SPI_READ);
        SPI.transfer(RXBNCTRL);
        for (byte i = 0; i < 14; i++)
        {
                if(DLCN==0) RxData[i]=0x00;
                else { RxData[i] = SPI.transfer(0x00); //to generate clock signal
to receive data over MISO
                if(i==5) DLCN=RxData[i];
                if(i>5) DLCN--;}
        }

        digitalWrite(CS, HIGH);

}

*/


//-------------------------------------------------------------

void CAN::RXNREAD(byte RXbuffer, StdFrame *Can_Msg)
{
  byte RXDATA[13];
  byte Address;

  if (RXbuffer == 0x01 || RXbuffer == 0x02)
  {
    if (RXbuffer == 0x01) Address = 0x61;
    else Address = 0x71;

    READ_RX_BUFFER(Address, &RXDATA[0], 13);

    Can_Msg[0].COB_ID = (unsigned int) ((RXDATA[0] << 3) | ((RXDATA[1] & 0xE0) >> 5));
    Can_Msg[0].RTR_BIT = (RXDATA[1] & 0x10) >> 4; // RXDATA[1] & b00010000(0x10)
Can_Msg[0].DLC = RXDATA[4] & 0x0F;

    for (byte i = 0; i < (Can_Msg[0].DLC); i++)
    {
      Can_Msg[0].DATA[i] = RXDATA[5 + i];
    }
  }
```

```
  if (RXbuffer == 0x03)
  {
    Address = 0x61;

    READ_RX_BUFFER(Address, &RXDATA[0], 13);

    Can_Msg[0].COB_ID = (unsigned int) ((RXDATA[0] << 3) | ((RXDATA[1] & 0xE0) >> 5));
    Can_Msg[0].RTR_BIT = (RXDATA[1] & 0x10) >> 4; // RXDATA[1] & b00010000(0x10)
Can_Msg[0].DLC = RXDATA[4] & 0x0F;

    for (byte i = 0; i < (Can_Msg[0].DLC); i++)
    {
      Can_Msg[0].DATA[i] = RXDATA[5 + i];
    }

    Address = 0x71;

    READ_RX_BUFFER(Address, &RXDATA[0], 13);

    Can_Msg[1].COB_ID = (unsigned int)(((RXDATA[0] << 8) | (RXDATA[1] & 0xE0)) >> 5);
    Can_Msg[1].RTR_BIT = (RXDATA[1] & 0x10) >> 4; // RXDATA[1] & b00010000(0x10)
Can_Msg[1].DLC = RXDATA[4] & 0x0F;

    for (byte i = 0; i < (Can_Msg[1].DLC); i++)
    {
      Can_Msg[1].DATA[i] = RXDATA[5 + i];
    }
  }

}

//-------------------------------------------------------------void CAN::WRITE(byte
Address, byte Data)
{ digitalWrite(CS, LOW);

  SPI.transfer(SPI_WRITE);
  SPI.transfer(Address);
  SPI.transfer(Data);

  digitalWrite(CS, HIGH);
}


//Funciona OKvoid CAN::WRITE_M(byte Address, byte DataArray[], byte Ndata)
{
  digitalWrite(CS, LOW);

  SPI.transfer(SPI_WRITE);
  SPI.transfer(Address);

  for (byte i = 0; i < Ndata; i++)
```

```
  {
    SPI.transfer(DataArray[i]);
  }

  digitalWrite(CS, HIGH);

}


//------------------------------------------------------------
/* CAN_LOAD_TX Buffer
Instruccion 01000abc + Dato 0000 0000
abc                 Address point     Address
000      0x0        TXB0SIDH      0x31
001      0x1        TXB0D0        0x66
010      0x2        TXB1SIDH      0x41
011      0x3        TXB1D0        0x46
100      0x4        TXB2SIDH      0x51
101      0x5        TXB2D0        0x56

Address Could Be  TXB0SIDH(0x31), RXB0D0(0x66), RXB1SIDH(0x71),RXB1D0(0x76)
the CAN_READ with address RXB0SIDH.. RXB1D0 function can replace CAN_READ_RX_Buffer but
with this function the message its shorter
*/
void CAN::LOAD_TX(byte Address, byte Data)
{
  byte LoadTxBuffer;
  switch (Address) {
    case TXB0SIDH:   LoadTxBuffer = SPI_WRITE_TX | 0x0; break;
    case TXB0D0:     LoadTxBuffer = SPI_WRITE_TX | 0x1; break;
    case TXB1SIDH:   LoadTxBuffer = SPI_WRITE_TX | 0x2; break;
    case TXB1D0:     LoadTxBuffer = SPI_WRITE_TX | 0x3; break;
    case TXB2SIDH:   LoadTxBuffer = SPI_WRITE_TX | 0x4; break;
    case TXB2D0:     LoadTxBuffer = SPI_WRITE_TX | 0x5; break;

    default:         LoadTxBuffer = SPI_WRITE_TX | 0x0; break;
  }


  digitalWrite(CS, LOW);

  SPI.transfer(LoadTxBuffer); //LOAD TX Buffer instruction  SPI.transfer(Data);
//Send DATA
  digitalWrite(CS, HIGH);
}


//------------------------------------------------------------
/*
Instruction controller to begin message transmission sequence for any of the transmit
buffers
TXB0 TXB1,TXB2
void CAN_RTS(byte TXn)   where TX could be char TXB0=0 "TXB1=1 TXB2=2
```

```
Instruction 10000nnn
n=1 TXB2  n=1 TXB1 n=1 TXB0
*/void CAN::RTS(byte TXBN)
{
  byte RTSTxn;

  switch (TXBN) {
    case TXB_0:   RTSTxn = SPI_RTS | 0x1; break;
    case TXB_1:   RTSTxn = SPI_RTS | 0x2; break;
    case TXB_2:   RTSTxn = SPI_RTS | 0x4; break;

    default:    RTSTxn = SPI_RTS | 0x1; break;
  }


  digitalWrite(CS, LOW);   //Habilita comunicacion con el mcp2515
  SPI.transfer(RTSTxn); //LOAD TX Buffer instruction
  digitalWrite(CS, HIGH); //deshabilita comunicacion con el mcp2515}

//-------------------------------------------------------------
/*
READ STATUS allow single instruction access to some of the often used status bits for
message reception and transmission
byte Data=CAN_READ_STATUS()   byte b7 b6 b5 b4 b3 b2 b1 b0
bit    Status bit

b0      CANINTF.RX0IF
b1      CANINTTFL.RX1IF
b2      TXB0CNTRL.TXREQ
b3      CANINTF.TX0IF
b4      TXB1CNTRL.TXREQ
b5      CANINTF.TX1IF
b6      TXB2CNTRL.TXREQ
b7      CANINTF.TX2IF

*/

byte CAN::READ_STATUS(void)
{
  byte Status;

  digitalWrite(CS, LOW);   //Habilita comunicacion con el mcp2515
  SPI.transfer(SPI_READ_STATUS); //SPI_READ_STATUS  instruction  Status =
SPI.transfer(0x00);  // STATUS DATA returned
  digitalWrite(CS, HIGH); //deshabilita comunicacion con el mcp2515
  return Status;

}

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

```
RX_STATUS INSTRUCTION This function it is used to get fast the RX Status Received message
MSG Type received and Filter MAtch
 Instruction 0xB0 Data received bits 786543210
 bits
 7 6  Received Message
 0 0 No Rx Message
 0 1 Message in RXB0
 1 0 Message in RXB1
 11 Message in both buffers

bits
4 3 Msg Type Received
0 0 Standrat Data frame
0 1 Standar remote frame
10 Extended data frame
11 Extended remote frame

bits
2 1 0 Filter MAtch
0 0 0 RXF0
0 0 1 RXF1
0 1 0 RXF2
0 1 1 RXF3
1 0 0 RXF4
1 0 1 RXF5
1 1 0 RXF0
1 1 1 RXF1
*/

byte CAN::RX_STATUS(void)
{
  byte Status;

  digitalWrite(CS, LOW);

  SPI.transfer(SPI_RX_STATUS);
  Status = SPI.transfer(0x00);

  digitalWrite(CS, HIGH);
  return Status;
}

//------------------------------------------------------------void
CAN::BIT_MODIFY(byte Address, byte Mask, byte Data)
{
  digitalWrite(CS, LOW);

  SPI.transfer(SPI_BIT_MODIFY);
  SPI.transfer(Address);
  SPI.transfer(Mask);
  SPI.transfer(Data);

  digitalWrite(CS, HIGH);
```

```
}
/*********************************************************************************
*****************************************************************
CAN OPERATION FUNCTIONS
These functios use the basic functions for SPI comunicacion with the MCP2515 and build
more complex functions
*********************************************************************************
*****************************************************************/
void CAN::SETMODE(char MODE[])
{
  byte ROMode;     //Request Operation mode  if (MODE == "NORMAL")
      { ROMode = 0x00;  }

  else if (MODE == "SLEEP")
      { ROMode = 0x20; } //0 0 1   0...0    =0x20
  else if (MODE == "LOOPBACK")
      { ROMode = 0x40;} //0 1 0   0...0    =0x40
  else if  (MODE == "LISTEN")
      { ROMode = 0x60;} //0 1 1   0...0    =0x60
  else if  (MODE == "CONFIGURATION")
      { ROMode = 0x80;} //1 0 0   0...0    =0x80
  else
      { ROMode = 0x00; }

  BIT_MODIFY(CANCTRL, 0xE0, ROMode);
}


/*****************************************************************/

String CAN::GETMODE(void)
{
  byte ROMode;
  String MODE;


  ROMode = READ(CANSTAT) & 0xE0;

  switch (ROMode) {
    case 0x00:    MODE = "NORMAL"; break;       //0 0 0   0...0    =0x0    case 0x20:
MODE = "SLEEP"; break;        //0 0 1   0...0    =0x20    case 0x40:    MODE =
"LOOPBACK"; break;     //0 1 0   0...0    =0x40    case 0x60:    MODE = "LISTEN";
break;       //0 1 1   0...0    =0x60    case 0x80:    MODE = "CONFIGURATION"; break;
//1 0 0   0...0    =0x80
    default:      MODE = "UNKNOW"; break; //By defect if ROMode did't match the
operation modes of the mcp2515  }

  return MODE;

}

/*****************************************************************/
```

```
void CAN::SPEEDCONFIG(int BAUDRATE)
{
  byte conf[3];

  // array configspeed{CNF3,CNF2,CNF1}  switch (BAUDRATE) {
    case 10:    conf[0] = 0x04; conf[1] = 0xB6; conf[2] = 0x27;        break;
    case 20:    conf[0] = 0x04; conf[1] = 0xb6; conf[2] = 0x13;        break;
    case 50:    conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x09;        break;
    case 100:   conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x04;        break;
    case 125:   conf[0] = 0x01; conf[1] = 0x91; conf[2] = 0x07;        break;
    case 250:   conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x01;        break;
    case 500:   conf[0] = 0x03; conf[1] = 0xAC; conf[2] = 0x00;        break;
    case 800:   conf[0] = 0x01; conf[1] = 0x9A; conf[2] = 0x00;        break;
    case 1000:  conf[0] = 0x01; conf[1] = 0x91; conf[2] = 0x00;        break;

  }

  WRITE_M(CNF3, conf, 3);

}



/*This is the same function of CAN_RX_STATUS the difference this return with pointers
the kind of message detected
bit
Received message                    Message type                          Filter
Match
7 6                                  4 3                                   2 1 0
0 0 No RxMessage                   0 0 Standar data Frame                0 0 0   RXF0
0 1 Message RXB0                   0 1 Standar remote frame              0 0 1   RXF1
1 0 Message RXB1                   1 0 Extended data Frame               0 1 0   RXF2
1 1 Message in both buffers        1 1 Extended remote frame             0 1 1   RXF3
                                                                          1 0 0
RXF4
                                                                          1 0 1
RXF5
                                                                          1 1 0
RXF0 Rollover to RXB1
                                                                          1 1 1
RXF1 Rollover to RXB1
*/
/*****************************************************************************
************/
void CAN::STATUSRXDATA(byte *RxBuffer, byte *MsgType, byte *FilterMatch)
{
  byte Status;

  Status = RX_STATUS();

  *RxBuffer = (Status & 0xC0) >> 6; //get the bits related with received message
*MsgType = (Status & 0x18) >> 3;
  *FilterMatch = (Status & 0x07);
```

```
}


/*****************************************************************************
***********/
/*Esta funcion limpia los bits recibidos en RXB0 y RXB1 para permitir que se reciban
nuevos datos
Address byte 0x2C CANINTF -Interrupt Flag        */
void CAN::CLEARBUFFERRXN(byte RXN)
{
  byte MASK;
  if(RXN==0) MASK=0x01;
  if(RXN==1) MASK=0x02;
  BIT_MODIFY(CANINTF, MASK, 0x00);
}

/*****************************************************************************
**********/
/*This function config the received buffer control RXB0
  RXB0CTRL  RXB1CTRL
  input binary number
  b00
  RXM1 RXM0
    1  1    Turn mask Filter OFF; receive any message
    1  0    Receive only extended message
    0  1    Receive only valid message with standart id
    0  0    Receive all valid message extended and standar id

  Rollover_enable   input b0  or b1
  BUKT

  1    RXB0 message will rover and be writen to RXB1 if RXB0 is full
  0    Rollover diable

*/

void CAN::RXBNCTRL_CONFIG(char RX_MODE[], char ROLLOVER_ENABLE)
{
  byte RXM;     //RXmode value  byte RXRE;    //RX Rollover enable bit  byte
RXBN_CONFIG = 0;

  if        (RX_MODE == "ANYM")  {
    RXM = B11;  //Config flag off receive any message  }
  else if   (RX_MODE == "EXTM")  {
    RXM = B10;  // Config receive only valid extended message  }
  else if   (RX_MODE == "STDM")  {
    RXM = B01;  // Config receive only valid standart message  }
  else if   (RX_MODE == "ALLVM") {
    RXM = B00;  // Config receive any valid extended or standar message  }
  else                          {
    RXM = B01;   //by deafault  receive valid standar message  }
```

```
  if      (ROLLOVER_ENABLE == 'E')    {
    RXRE = 0X01; //Rollover Enable  }
  else if (ROLLOVER_ENABLE == 'D')    {
    RXRE = 0x00; //Rollover disbale  }
  else                               {
    RXRE = 0x01; //by defect its Enable  }


  RXM = RXM << 5;
  RXRE = RXRE << 3;
  RXBN_CONFIG = RXM & RXRE;

  BIT_MODIFY(RXB0CTRL, 0x64, RXBN_CONFIG); // MASK 0x64 B01100100  RXBN_CONFIG = RXM;
  BIT_MODIFY(RXB1CTRL, 0x60, RXBN_CONFIG);

}

/*****************************************************************************
*******/

/*
This function config the MASK RXB0 and RXB1  This function only implement MASK for
STANDARD message
RXMASK, MASK its a 11 bits ID it can take values from 0 to 0X7FF as maximum value allouded

input RXMASKN = RXM0SIDH(0x20)  or RXM1SIDH (0x24)
*/
void CAN::SETMASK_RXMN(byte RXMASKN, unsigned int MASK_HEX)
{
  byte MASK_ID[2];

  MASK_ID[0] = byte ((MASK_HEX & 0x7F8) >> 3);
  MASK_ID[1] = byte ((MASK_HEX & 0x007) << 5);
    // CAN_WRITE_M(RXMASKN,MASK_ID,2);  //Write over RXMnSIDH and RXMnSIDL register at
same time.  WRITE_M(RXMASKN, MASK_ID,2);

}


/*****************************************************************************
******/
/* This function only works when the mcp2515 it's on configuration mode. */

void CAN::SETFILTER_RXFN(byte RXFILTERN, unsigned int FILTER_HEX)
  {
    byte FILTER_ID[2];

    FILTER_ID[0]=byte ((FILTER_HEX & 0x7F8) >> 3);   //GET SIDH    FILTER_ID[1]=byte
((FILTER_HEX & 0x007) << 5);   //GET SIDL   WRITE_M(RXFILTERN,FILTER_ID,2);  //Write
over RXMnSIDH and RXMnSIDL register at same time.
   }
```

```
void CAN::LOADTX(byte TXBN, StdFrame CanMsg)
{
  byte TXAddress, IDH, IDL, DLCbyte;  //Priority value  byte DTX[14] = {0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}; // TXBNCTRL,SIDH,
SIDL, EID8 (0x00),EID0(0x00),DLC, TXBnDM
  // const byte Mask=0x03;   //Mask to modify TXBNCTRL for priority configuration of
the message to transmit
  //    if (Priority=="_LOWEST")        PrVal=0x00;
  //    else if(Priority=="_LOW")       PrVal=0x01;
  //    else if(Priority=="_HIGH")      PrVal=0x02;
  //    else if(Priority=="_HIGHEST")   PrVal=0x03;
  //    else PrVal=0x03;     //by default is highest priority value

  IDH = byte((CanMsg.COB_ID & 0x7F8) >> 3); //Get byte  SIDH from COB_ID  IDL =
byte((CanMsg.COB_ID & 0x007) << 5);   //Get byte SIDL from COB_ID  DLCbyte =
( CanMsg.RTR_BIT  << 6 ) | CanMsg.DLC;

  TXAddress = TXBN; //TXBN;    //TXBNCTRL pointer
  //DTX[0]=0x03;  //TXBnCTRL  DTX[1] = IDH;  //SIDH  DTX[2] = IDL;  //SIDL  DTX[3] =
0x00; //EID8  DTX[4] = 0x00; //EID0  DTX[5] = DLCbyte;

  for (byte i = 0; i < CanMsg.DLC; i++)
  {
    DTX[6 + i] = CanMsg.DATA[i];
  }



  for (byte i = 0; i < 14; i++)
  {
    WRITE(TXAddress, DTX[i]);
    TXAddress++;
  }

}

/*****************************************************************************
*******************/


byte CAN::SENDTXBn(byte TXBN)
{
  byte TX_Transmit;    //0 Send TXBN sucesfull   1 Fail to send TXBN  byte TXSTATUS,
i = 0;


  RTS(TXBN);

  delay(10);
```

```
  while (TXSTATUS == 1)
  {
    TXSTATUS = (READ(TXBN) & 0x78) >> 3;
    i++;
    if (i == 100) break;
  }

  if (i == 100)  TX_Transmit = 1;  // if transmir fail return TX_Transmit 1  else
TX_Transmit = 0;          // if transmit sucesfull TX_Transmit =0
  return (TX_Transmit);

}

/*****************************************************************************
*****************************************///// CAN OPEN FUNCTIONS
CANOPEN_NMTSERVICE(char NMTC[], byte node)void CAN::NMTSERVICE(char NMTC[],byte Node)
{
  StdFrame CANTX;
  CANTX.COB_ID=0;
  CANTX.RTR_BIT=0;
  CANTX.DLC=0x02;
  CANTX.DATA[2]; //={0x00,0x00};

   if (NMTC== "OPERATIONAL")               CANTX.DATA[0]=0x01;
   else if (NMTC=="STOP")                  CANTX.DATA[0]=0x02;
   else if (NMTC=="PRE-OPERATIONAL")       CANTX.DATA[0]=0x80;
   else if (NMTC=="RESET-NODE")            CANTX.DATA[0]=0x81;
   else if (NMTC=="RESET-COMMUNICATION")   CANTX.DATA[0]=0x82;

   CANTX.DATA[1]=Node;

  LOADTX(TXB_0,CANTX);
  SENDTXBn(TXB_0);

 }


/*****************************************************************************
*****************************************************
   Turn OFF for 2 seconds the CAN device conected  during two seconds and wait to seconds
after the rele has turned ON again
   PINRELE (Pin to control the rele) is declared as global constant var on
CANGlobalVArs.h PIN 7 but it could be nodified if needed
*****************************************************************************
*****************************************************/

 void CAN::ONOFF(void)
  {  pinMode(PINRELE, OUTPUT);           // set pin as output
digitalWrite(PINRELE, HIGH);        //Turn OFF CAN Device     delay(2000);
//Wait two seconds     digitalWrite(PINRELE,LOW);              //Turn ON  CAN Device
delay(2000);                        //Wait two seconds     }
```

```
/*****************************************************************************
*************************************************************
Auto baud rate detect  This function detect the baud rate of the sensor automatically,
to get the baud rate it do the following in a cycle for testing all the baud the possible
baud rates.
1. Reset mcp2515
2. SET mcp2515 in configuration mode
3. config baut rate (1000 800 500 250 125 100 50 20 10)  one by one in each cylce
4. config the mcp2515 to receive standard message only
5. enable all interrupt from register CANINTE
6. clear interrupt register CANINTF
7. SET mcp2515 on listen mode
8. ON/OFF sensor
9 if the interrupt MERRF (Message error interrupt flag its ON it mean there is not valid
message but it the interrupt is OFF then it mean a valid message has been received and
the baud rate has been detected

*****************************************************************************
*************************************************************/
int CAN::AUTO_BAUD_DETECT(void)
{
         const int VSpeed[9]={1000,800,500,250,125,100,50,20,10};
         byte RXBuffer,Status;
        byte i;
         StdFrame RXDATA;

         for(i=0; i<9; i++)
         {
                  RESET();
                  delay(100);
                  SETMODE("CONFIGURATION");
                  SPEEDCONFIG(VSpeed[i]);
                  RXBNCTRL_CONFIG("STDM", 'E');
                  BIT_MODIFY(CANINTE,0xFF,0xFF);  //Interrupt enable
         BIT_MODIFY(CANINTF,0xFF,0x00);  //clear interrupt register
         SETMODE("LISTEN");              //Set Listen Mode
                  if(i==0) {ONOFF();}

                  for(byte j=0;j<4;j++)     //check data at least 8 times at time          {
                           Status=WAITRXN(&RXDATA,&RXDATA);
//RXNREAD(0x01,&RXDATA);                        RXBuffer=READ(CANINTF);
                        RXBuffer=RXBuffer & 0x80;
                           if (RXBuffer==0) break;
                  }

                  if(RXBuffer==0) break;
         }


         SETMODE("NORMAL");
```

```
        if(RXBuffer==0)  return VSpeed[i];    //if RXBuffer ==0 means the speed has
been detected and return the last value, if RXBuffer !=0 then retunr 0 that means the
speed detecting has fail.   else return 0;

}


/**************************************************************************************
***
 This function wait for a data in the buffer and dependig if the data is get it on RXB0
or RXB1 ot in both, load the RXBN and return the status to indicate where the data has
been received.
 return status
 00 no data load
 01 Data in RXB0
 10 Data in RXB1
 11 Data in RXB0 and RXB1

Timeout this value its the maximum time waiting for a data

**************************************************************************************
**/
byte CAN::WAITRXN(StdFrame *CAN_RXB0,StdFrame *CAN_RXB1)
{
  byte Status;
  StdFrame FrameBuffer;
  int timeout=1000;

  for(int i=0; i<timeout; i++)
  {  Status=STATUSRX();
     if(Status != 0) break;
     delay(1);
   }

   if(Status==0) return Status;
   else if(Status==1){ RXNREAD(0x01,CAN_RXB0); return Status;}
//CLEARBUFFERRXN();  Clear CANINTF to deleate all register related with received data
else if(Status==2){ RXNREAD(0x02,CAN_RXB1);  return Status;}
   else if(Status==3){ RXNREAD(0x01,CAN_RXB0); RXNREAD(0x02,CAN_RXB1); return
Status;}

 }



  /*******************************************************
  This function detect if a message has been received it check the bit 7 and 6
  7  6 Received Message
  0  0 No Rx message
  0  1 Message in RXB0
  1  0 Message in RXB1
  1  1 Message in both buffers
```

```
  Retunr byte Status 0 No message 1 Message on RXB0, 2 Message on RXB1, 3 Message on
both buffers.

  ************************************************************/


  byte CAN::STATUSRX (void)
  {
          byte Status;

          Status=RX_STATUS();
          Status=Status & 0xC0;    //clear unused bits on the status byte register.
          Status=Status >> 6;
          return Status;
  }


/*************************************************************************************
**************************
  Node Detect, this function send a node guarding COB-ID 80 0 0 0 0 0 0 0 0 0 and wait
the answer of the sensor
  take the COB-ID and determin the Node


*************************************************************************************
**************************/
  int CAN::NODE_DETECT(void)
{
  int NODE,RestaNodo;
  byte Status;
  StdFrame NGuarding,CANRXB0,CANRXB1;

  /* Node Guarding
  COB-ID  RTR  DLC DATA
  080      0    0  0 0 0 0 0 0 0
    */
  NGuarding.COB_ID=0x80;
  NGuarding.RTR_BIT=0;
  NGuarding.DLC=0;
  NGuarding.DATA[1]=0;

 // SEND Node Guarding
  NMTSERVICE("OPERATIONAL",0x00);
  LOADTX(TXB0,NGuarding);
  SENDTXBn(TXB0);

  Status=WAITRXN(&CANRXB0,&CANRXB1);

  if(Status==0) return NODE=0;
  else if(Status==1||Status==3)     //If there are a message on RXB0 or in both register
read COB_ID over RXB0            { if((CANRXB0.COB_ID>=0x701) && (CANRXB0.COB_ID <=
0x7FF)) RestaNodo=0x700;
```

```
              else  RestaNodo=(CANRXB0.COB_ID & 0x0700) | 0x080;
              NODE=CANRXB0.COB_ID-RestaNodo;}

  else if(Status==2)
          {  if((CANRXB1.COB_ID>=0x701) || (CANRXB0.COB_ID <= 0x7FF))
RestaNodo=0x700;
              else  RestaNodo=(CANRXB0.COB_ID & 0x0700) | 0x080;
              NODE=CANRXB0.COB_ID-RestaNodo;  }
  return NODE;
  }
```

**********MAIN PROGRAM**********

```
#include <CANMEGA.h>
#include <CANMEGAGlobalVars.h>
#include <mcp2515_defs.h>
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include <RTClib.h>

CAN CAN;
RTC_DS3231 rtc;    // Real time circuit DS3231

//GLOBAL VARS

byte FRX0=0,FRX1=0;      //Flags for interrupt
int Index,MaxIndex,LastIndex,KeyEnter;    //Vars for key menu handle
int RXMTS1,RXMTS2,RXMTS3,RXAT1,RXAT2,RXAT3,RXP1,RXP2,RXP3,TXN;
StdFrame BRX0,BRX1;


String Menu_set[14]={     };
String Menu_clr[14]={     };
String Menu_Enter[14]= {    };


//Datalogger Vars
unsigned long CTIME,CTIME_OLD;       //   day    month    hour    Min   //   00      00       00     00
Current time updated in every cicle.
int IW0=0,IW0old=0;      // index write.
const int Bmax=100;      //Buffer maximo
File DataFile;
typedef struct { byte N;      //Error number N
                        byte ErrorID;
                        int ErrorCode;                //   day    month    hour    Min
                        unsigned long rtc;            //   00      00       00     00
                       }   ME_type;

ME_type MsgError[Bmax];
```

```
void setup() {

   CAN.CANMEGA_CONFIG();
   Serial2.begin(115200);
   Serial.begin(115200);
   rtc.begin();

      if(!SD.begin(CS_SDCARD)) { Serial.println("SDCARD FAIL"); }      //SD card file init Fail
   delay(2000);
   NEXTION("page Menu");

   }

void loop() {
   // put your main code here, to run repeatedly:
   MainMenu();

}



void MainMenu()
{    detachInterrupt(digitalPinToInterrupt(INTKEY));

     //INIT MENU
     Index=0;
     MaxIndex=4;
     KeyEnter=0;
     LastIndex=0;

     // Init Menu list
       Menu_set[0]=    "b0.bco=14";    //Config Sensor //594
       Menu_set[1]=    "b1.bco=14";    // Test Line
       Menu_set[2]=    "b2.bco=14";    // Datalogger
       Menu_set[3]=    "b3.bco=14";    // Read Data
       Menu_set[4]=    "b4.bco=14";    // Settings

       Menu_clr[0]= "b0.bco=3135";    //Config Sensor
       Menu_clr[1]= "b1.bco=3135";    // Test Line
       Menu_clr[2]= "b2.bco=3135";    // Datalogger
       Menu_clr[3]= "b3.bco=3135";    // Read Data
       Menu_clr[4]= "b4.bco=3135";    // Settings

       Menu_Enter[0]= "click b0,1"; //Config Sensor
       Menu_Enter[1]= "click b1,1"; // Test Line
       Menu_Enter[2]= "click b2,1"; // Datalogger
       Menu_Enter[3]= "click b3,1"; // Read Data
       Menu_Enter[4]= "click b4,1"; // Settings

       NEXTION(Menu_set[0]); //Init button 0 selected
       attachInterrupt(digitalPinToInterrupt(INTKEY), KEYBOARD_MENU,FALLING );
   while(1)    {

       delay(100);
       displayRTC();
       if(KeyEnter==1) {
              switch(Index)
```

```
                {     case 0: {NEXTION("page Configuration"); ConfigurationMenu();    break;}
                      case 1: {NEXTION("page GlobalTest");       GlobalTest();      break;}
                      case 2: {NEXTION("page Logger");            DataLogger();      break;}
                      case 3: {NEXTION("page Logger");            ReadLogger();      break;}
                      case 4: {NEXTION("page Setting");           Settings();        break;}
                         }

                         KeyEnter=0; break;
               } }


******************CANOPEN *********************
// CAN OPEN protocol implemented

//*************** LAYER SETTING SERVICE ************************


  byte    LSSCONFIGURATION(int Node,int BR)    // Node, BR= Baud Rate
        {         // int timeout=1000;

                  StdFrame RXBUFFER;
                  StdFrame    LSS_STOP=                                          {0x7E5,    0x00,    0x08,
{0x04,0x01,0x00,0x00,0x00,0x00,0x00,0x00}};     //SET Configuration State
                  StdFrame     LSS_TXSETNODE=                                  {0x7E5,    0x00,    0x08,
{0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x00}};     //CONFIG NODE NODEmust be inserted in DATA 01
                  StdFrame     LSS_RXSETNODE=                                  {0x7E4,    0x00,    0x08,
{0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x00}};     //Sensor Answer if everything its OK
                  StdFrame        LSS_TXSETBaudRate=                      {0x7E5,        0x00,        0x08,
{0x13,0x00,0x00,0x00,0x00,0x00,0x00,0x00}};   //Config Baud rate to 250,    Data 03=03 for 250
                  StdFrame        LSS_RXSETBaudRate=                      {0x7E4,        0x00,        0x08,
{0x13,0x00,0x00,0x00,0x00,0x00,0x00,0x00}}; //Sensorr Answer if everything its OK
                  StdFrame    LSS_TXSAVE=                                        {0x7E5,    0x00,    0x08,
{0x17,0x00,0x00,0x00,0x00,0x00,0x00,0x00}};
                  StdFrame    LSS_RXSAVE=                                        {0x7E4,    0x00,    0x08,
{0x17,0x00,0x00,0x00,0x00,0x00,0x00,0x00}};
                  StdFrame    LSS_NORMAL=                                        {0x7E5,    0x00,    0x08,
{0x04,0x00,0x00,0x00,0x00,0x00,0x00,0x00}};     //SET SENSOR in NORMAL MODE

                  LSS_TXSETNODE.DATA[1]=Node;


                //BR variable is going to be reused to assign the value accordingly to the table 0 of the
CAN open standar Baud.
                  switch(BR){
                                case 1000: BR=0; break;
                                case    850: BR=1; break;
                                case    500: BR=2; break;
                                case    250: BR=3; break;
                                case    125: BR=4; break;
                                case     50: BR=6; break;
                                case     20: BR=7; break;
                                case     10: BR=8; break;
                                case       0: BR=9; break;    //Automatic detection if the sensor support
this config.
                             }

                  LSS_TXSETBaudRate.DATA[2]=BR;
```

```
                CAN.NMTSERVICE(NMT_STOP,0); delay(100);    //STOP SENSOR

                CAN.SENDTXBn(TXB_0, LSS_STOP); delay(100);

                //***************SET BAUD NODE
                CAN.SENDTXBn(TXB_0,LSS_TXSETNODE); delay(100);

                 if(WaitAnswer()==0) {Serial.println("FAIL LSS SET NODE"); return 0;}
                 if(FRX0==1)      {FRX0=0;  if(BRX0.COB_ID==0x7E4)  RXBUFFER=BRX0;  }      //Clear  Flag
FRX0 or FRX1
                 if(FRX1==1)    {FRX1=0; if(BRX1.COB_ID==0x7E4) RXBUFFER=BRX1; }

                    //PRINTFRAME(RXBUFFER);
                    //PRINTFRAME(LSS_RXSETNODE);

                    if(memcmp(&RXBUFFER,&LSS_RXSETNODE,sizeof(RXBUFFER)))
{Serial.println("FAIL LSS SET NODE Comparacion"); return 0;}    // memcmp its 0 if the struct are equal


                  //***************SET BAUD RATE
                    CAN.SENDTXBn(TXB_0,LSS_TXSETBaudRate); delay(100);

                    if(WaitAnswer()==0) {Serial.println("FAIL SET BR"); return 0;}
                    if(FRX0==1)     {FRX0=0;     if(BRX0.COB_ID==0x7E4)  RXBUFFER=BRX0;  }     //Clear
Flag FRX0 or FRX1
                    if(FRX1==1)    {FRX1=0;    if(BRX1.COB_ID==0x7E4) RXBUFFER=BRX1; }

                    if(memcmp(&RXBUFFER,&LSS_RXSETBaudRate,sizeof(RXBUFFER)))
{Serial.println("LSS answer FAIL SET BR"); return 0;}    // memcmp its 0 if the struct are equal


                  //********** Save Configuration ********

                    CAN.SENDTXBn(TXB_0,LSS_TXSAVE);    delay(100);

                    if(WaitAnswer()==0) {Serial.println("FAIL LSS SAVE SETTINGS"); return 0;}
                    if(FRX0==1)     {FRX0=0;  if(BRX0.COB_ID==0x7E4)RXBUFFER=BRX0;  }     //Clear  Flag
FRX0 or FRX1
                    if(FRX1==1)    {FRX1=0; if(BRX1.COB_ID==0x7E4)RXBUFFER=BRX1; }

                    if(memcmp(&RXBUFFER,&LSS_RXSAVE,sizeof(RXBUFFER)))        {Serial.println("LSS
ANSWER FAIL SAVE "); return 0;}    // memcmp its 0 if the struct are equal

                    CAN.SENDTXBn(TXB_0,LSS_NORMAL); delay(200);
                    Serial.println("LLS COMPLETO");
                    return 1;
              }



  /****************** SDO SERVICE ********************************

    SDOWRITE FUNCTION
    *Input parameters*
    Node:          Number of node
    Index:         Index from dictionary
```

```
    Subindex:    Subindex.
    DATA:          Data to write

    Return byte 1 -if the Data has been writed correctly and 0 if fail to write Data.
  **********************************************************************/


 byte SDOWRITE(byte Node,int Index,int SubIndex, long DATA)
  {
     StdFrame RXBUFFER;
     StdFrame TXBUFFER;
     byte LIndex;
     byte HIndex;
     byte D0,D1,D2,D3;      // Data byte extraction
     int timeout=1000;      //timeout for waiting an answer before to return 0

     LIndex= byte (((Index & 0x000FF)));
     HIndex= byte ((Index & 0xFF00)>>8);
     D0= byte( DATA & 0x000000FF);
     D1= byte((DATA & 0x0000FF00)>>8);
     D2= byte((DATA & 0x00FF0000)>>16);
     D3= byte((DATA & 0xFF000000)>>24);

     TXBUFFER.COB_ID=0x600+Node;
     TXBUFFER.RTR_BIT=0;
     TXBUFFER.DLC=8;
     TXBUFFER.DATA[0]=0x22;          // 0x22 Write SDO Command
     TXBUFFER.DATA[1]=LIndex;      //LIndex for Index dictionariy object
     TXBUFFER.DATA[2]=HIndex;      // HIndex for index dictionary object
     TXBUFFER.DATA[3]=SubIndex;
     TXBUFFER.DATA[4]=D0;
     TXBUFFER.DATA[5]=D1;
     TXBUFFER.DATA[6]=D2;
     TXBUFFER.DATA[7]=D3;


   //Send Data
     CAN.SENDTXBn(TXB_0,TXBUFFER); delay(500);


     if(WaitAnswer()==0) return 0;
     if(FRX0==1) {    FRX0=0; if(BRX0.COB_ID==0x580+Node) RXBUFFER=BRX0; }

     if(FRX1==1){ FRX1=0; if(BRX1.COB_ID==0x580+Node) RXBUFFER=BRX1; }



     if(RXBUFFER.COB_ID==0x580+Node              &&              RXBUFFER.DATA[0]==0x60              &&
RXBUFFER.DATA[1]==LIndex && RXBUFFER.DATA[2]==HIndex && RXBUFFER.DATA[3]==SubIndex)
                  {return 1;    }

     else            {Serial.println("Noigual"); return 0;}

}
```

```
/*****************************************************************
 * SDO READ
 * NODE, Index
 *  return long Data  return 0 if reading fail  Check what happen if data
=0x000000  ??????????????????
 *
 */
long SDOREAD(byte Node,int Index,byte SubIndex)
  {
    StdFrame RXBUFFER;
    StdFrame TXBUFFER;
    byte LIndex;
    byte HIndex;
    byte D0,D1,D2,D3;    // Data byte extraction
    int timeout=1000;    //timeout for waiting an answer before to return 0
    long Data;

    CAN.CLEARBUFFERRXN(2);//Clear Buffer

    LIndex= (Index & 0x000FF);
    HIndex= (Index & 0xFF00)>>8;

    TXBUFFER.COB_ID=0x600+Node;
    TXBUFFER.DLC=8;
    TXBUFFER.DATA[0]=0x40;       // 0x40 Read SDO Command
    TXBUFFER.DATA[1]=LIndex;    //LIndex for Index dictionariy object
    TXBUFFER.DATA[2]=HIndex;    // HIndex for index dictionary object
    TXBUFFER.DATA[3]=SubIndex;
    TXBUFFER.DATA[4]=0;
    TXBUFFER.DATA[5]=0;
    TXBUFFER.DATA[6]=0;
    TXBUFFER.DATA[7]=0;


  //Send Data
    CAN.SENDTXBn(TXB_0,TXBUFFER);

//      while(timeout){    delay(1);
//                         if((FRX0 || FRX1) == 1) break;
//                         else timeout--;
//                      }
    Serial.print("Timeout="); Serial.println(timeout);
    if(WaitAnswer()==0) return 0;
    else if(FRX0==1) { FRX0=0,   RXBUFFER=BRX0;}
    else if(FRX1==1) { FRX1=0;   RXBUFFER=BRX1;}

    if(RXBUFFER.COB_ID==0x580+Node  &&  (RXBUFFER.DATA[0]&0xF0)==0x40  &&
RXBUFFER.DATA[1]==LIndex && RXBUFFER.DATA[2]==HIndex && RXBUFFER.DATA[3]==SubIndex)
                { D0=RXBUFFER.DATA[4];
                  D1=RXBUFFER.DATA[5];
                  D2=RXBUFFER.DATA[6];
                  D3=RXBUFFER.DATA[7];
                  Data=long(D0) | long(D1)<<8 | long(D2)<<16 | long(D3)<<24;
                  return Data;
```

```
                        }

      else              {Serial.println("Noigual"); return 0;}

}


byte WaitAnswer()
{         int timeout=2000;

          while(timeout)
                {      delay(1);
                       if((FRX0 || FRX1) == 1) break;
                       else timeout--;
                          }
          if(timeout==0) return 0;
          else return(1);

}

void PRINTFRAME(StdFrame Buffer)
{
Serial.print(Buffer.COB_ID,HEX); Serial.print(" ");
Serial.print(Buffer.DLC,HEX); Serial.print(" ");
for(byte i=0;i<8;i++)
{ Serial.print(Buffer.DATA[i],HEX);
   }
   Serial.println(" ");
}
/*
byte CompareCANMSG(StdFrame CANMSG1, StdFrame CANMSG2)
{
     if(CANMSG1.COB_ID==CANMSG2.COB_ID) COM_COBID=1;
     else COM_COBID=0;
     if(


}


*******CONFIG MENU *******

void ConfigurationMenu(void)
 {
       Index = 2;
       MaxIndex=10;
       KeyEnter=0;          //State or Enter key
       LastIndex=0;

       Menu_set[0]= "b0.bco=594";          // Button Back
       Menu_set[1]= "b1.bco=594";          //DETECT
       Menu_set[2]= "b2.bco=31695";        //MTS 1
       Menu_set[3]= "b3.bco=31695";        //MTS 2
       Menu_set[4]= "b4.bco=31695";        //MTS 3
       Menu_set[5]= "b5.bco=31695";        //Atos 1
       Menu_set[6]= "b6.bco=31695";        //Atos 2
       Menu_set[7]= "b7.bco=31695";        //Atos 3
```

```
    Menu_set[8]= "b8.bco=31695";       //Trafag 1
    Menu_set[9]= "b9.bco=31695";       //Trafag 2
    Menu_set[10]= "b10.bco=31695";    //Trafag 3

    Menu_clr[0]= "b0.bco=3135";         // Button Back
    Menu_clr[1]= "b1.bco=3135";         //DETECT
    Menu_clr[2]= "b2.bco=65535";       //MTS 1
    Menu_clr[3]= "b3.bco=65535";       //MTS 2
    Menu_clr[4]= "b4.bco=65535";       //MTS 3
    Menu_clr[5]= "b5.bco=65535";       //Atos 1
    Menu_clr[6]= "b6.bco=65535";       //Atos 2
    Menu_clr[7]= "b7.bco=65535";       //Atos 3
    Menu_clr[8]= "b8.bco=65535";       //Trafag 1
    Menu_clr[9]= "b9.bco=65535";       //Trafag 2
    Menu_clr[10]= "b10.bco=65535";    //Trafag 3


    Menu_Enter[0]=    "click b0,1";     // Button Back
    Menu_Enter[1]=    "click b1,1";     //DETECT
    Menu_Enter[2]=    "click b2,1";     //MTS 1
    Menu_Enter[3]=    "click b3,1";     //MTS 2
    Menu_Enter[4]=    "click b4,1";     //MTS 3
    Menu_Enter[5]=    "click b5,1";     //Atos 1
    Menu_Enter[6]=    "click b6,1";     //Atos 2
    Menu_Enter[7]=    "click b7,1";     //Atos 3
    Menu_Enter[8]=    "click b8,1";     //Trafag 1
    Menu_Enter[9]=    "click b9,1";     //Trafag 2
    Menu_Enter[10]= "click b10,1";    //Trafag 3

    String SensorFile [11]=          {    "Nofile",
                                              "Nofile",
                                              "MTS1.dcf",
                                              "MTS2.dcf",
                                              "MTS3.dcf",
                                              "ATOS1.dcf",
                                              "ATOS2.dcf",
                                              "ATOS3.dcf",
                                              "TRAFAG1.dcf",
                                              "TRAFAG2.dcf",
                                              "TRAFAG3.dcf",
                                         };


       NEXTION(Menu_set[2]); //Init button 0 selected
       attachInterrupt(digitalPinToInterrupt(INTKEY),KEYBOARD_MENU,FALLING );

    while(1)
    {
                            if(KeyEnter==1&&Index==0)    //BACK to Main menu
                                    { KeyEnter=0;
                                       detachInterrupt(digitalPinToInterrupt(INTKEY));
                                       Index=0; break; }    // NEXTION("page Menu"); return;    }

                            else if(KeyEnter==1&&Index==1){ //Detect Menu

detachInterrupt(digitalPinToInterrupt(INTKEY));
                                                                KeyEnter=0;
```

```
                                                                DETECT(); NEXTION(Menu_set[1]);
//                                                                Serial.print("Index:");
//                                                                Serial.println(Index);
//                                                                Serial.print("KeyEnter:");
//                                                                Serial.println(KeyEnter);

attachInterrupt(digitalPinToInterrupt(INTKEY), KEYBOARD_MENU,FALLING );}


                        else if(KeyEnter==1&&Index>=2)
                                { detachInterrupt(digitalPinToInterrupt(INTKEY));
                                  SensorDataFile(SensorFile[Index]);    //Read Data from file
                                  attachInterrupt(digitalPinToInterrupt(INTKEY),
KEYBOARD_MENU,FALLING );
                                  while(KeyEnter>0)
                                  {         if(KeyEnter==2)
                                            {    detachInterrupt(digitalPinToInterrupt(INTKEY));
                                                 CONFIG_SENSOR(SensorFile[Index]);
                                                 //Serial.println("Config Sensor file:");
                                                   //Serial.println(SensorFile[Index]);
                                                attachInterrupt(digitalPinToInterrupt(INTKEY),
KEYBOARD_MENU,FALLING );
                                                 KeyEnter=1;
                                             }
                                             delay(50);
                                  }


                                  }


      }
      NEXTION("page Menu");
      Serial.println("Exit Confg Mennu");
 }


**********Config Sensor********
/CONFIG SENSOR


byte CONFIG_SENSOR(String SensorFile)
   {     detachInterrupt(digitalPinToInterrupt(INTKEY));

      if(digitalRead(SDDETECTED)==1)          //if SD card is inserted continue else display NO SD
detected.
      {
            int CurrentNode=0,CurrentBaud=0;
            byte FAIL=0;
            char FileName[SensorFile.length()+1];

            SensorFile.toCharArray(FileName,sizeof(FileName));

            File myFile;
            String TXString_Line,Displaytxt;
            char   CHDevice[14],CHBaudRate[5],CHNode[5],CHSensorType[10],CHSDODATA[16];   //char
array to store the byte before to turn it on a     HEXADECIMAL byte number
```

```
  int Index,SubIndex,NewNode,NewBaud;    //Data from file
  long Data;    //Data from file
 INT_CANREAD_OFF();

 NEXTIONTXT("INFO","DETECTING BAUD RATE");

 CurrentBaud=CAN.AUTO_BAUD_DETECT();
 Serial.println(CurrentBaud);
 NEXTIONTXT("BRSENSOR",(String)CurrentBaud);

 //If sensor baud rate has not been detected cancel the sensor configuration
 if(CurrentBaud==0)  {NEXTIONTXT("NSENSOR","0");  NEXTIONTXT("INFO","FAIL SENSOR NO
DETECTED"); return 0;}
 else if(CurrentBaud!=0)
{
        NEXTIONTXT("INFO","Detecting Node");
        CurrentNode=CAN.NODE_DETECT();
        Serial.println(CurrentNode);
        NEXTIONTXT("NSENSOR",(String)CurrentNode);

          myFile = SD.open(FileName,FILE_READ);      //configuration file name

          if (myFile) {          TXString_Line=myFile.readStringUntil('\n');
                                      TXString_Line.toCharArray(CHDevice,14);

                                      TXString_Line=myFile.readStringUntil('\n');
                                      TXString_Line.toCharArray(CHBaudRate,5);
                                      NewBaud=int(strtoul(CHBaudRate,NULL,10));

                                      TXString_Line=myFile.readStringUntil('\n');
                                      TXString_Line.toCharArray(CHNode,5);
                                      NewNode=int(strtoul(CHNode,NULL,10));


                                      //TXString_Line=myFile.readStringUntil('\n');
                                      //TXString_Line.toCharArray(SensorType,12);

//                                        NEXTIONTXT("INFO","CONECT IXXAT");
//                                        delay(10000);
                                     NEXTIONTXT("INFO","Programing LSS");
                                     delay(10);
                                     Serial.println("Starting LSS");
                                     Serial.print("NEW NODE: "); Serial.println(NewNode);
                                     Serial.print("NEW BAUD: "); Serial.println(NewBaud);

                                        CAN.NMTSERVICE(NMT_STOP,0); delay(100);
                                        CAN.CLEARBUFFERRXN(2);          delay(100);

                                        INT_CANREAD_ON();       //Turn ON interrrupr for CAN
reading

                                        LSSCONFIGURATION(NewNode,NewBaud);

                                        INT_OFF();

                                    CurrentBaud=CAN.AUTO_BAUD_DETECT();
```

```
                                        Serial.print("NewBaud config: ");
                                        Serial.println(CurrentBaud);

                                         CurrentNode=CAN.NODE_DETECT();
                                         Serial.print("NewNode detected: ");
                                        Serial.println(CurrentNode);

                                        CAN.NMTSERVICE(NMT_RESETNODE,0);
                                        CAN.NMTSERVICE(NMT_STOP,0);
                                        CurrentNode=NewNode;

                                        NEXTIONTXT("INFO","New BR and Node OK");
                                        NEXTIONTXT("NSENSOR",(String)NewNode);
                                        NEXTIONTXT("BRSENSOR",(String)NewBaud);
                                        NEXTIONTXT("INFO","Programing SDO");
                                        CAN.CLEARBUFFERRXN(2);delay(100);


                                while(myFile.available())
                                    {           NEXTIONTXT("INFO","Programing SDO.");
                                                delay(100);
                                                //Read the line
                                                TXString_Line = myFile.readStringUntil('\n');
                                                TXString_Line.toCharArray(CHSDODATA,17);

                                                NEXTIONTXT("b11","Programing SDO..");
                                                delay(100);

SDOSTRING_TO_SDONUM(CHSDODATA,&Index,&SubIndex,&Data);
                                                NEXTIONTXT("INFO","Programing SDO...");
                                                delay(100);


                                                if(SDOWRITE(CurrentNode,Index,SubIndex,Data)==0)
{Serial.println("FailSDO Configuration");FAIL=1;    break;}
                                            }
                                                CAN.NMTSERVICE(NMT_RESETNODE,0);
                                                NEXTIONTXT("INFO","CONFIGURATION OK");
                                                        }
                    else {
                            // if the file didn't open, print an error:
                           Serial.println("error opening test.txt");
                           NEXTIONTXT("INFO","FILE NOT FOUND");
                           FAIL=1;
                            }
                //   }
                        myFile.close();
                        if(FAIL==1)         {  NEXTIONTXT("INFO","FAIL");  Serial.println("Configuration   Fail");
return 0;}
                        else                { Serial.println("Configuration OK");       return 1;}
                        INT_CANREAD_OFF();

                     }

               else NEXTIONTXT("INFO","SD CARD NOT DETECTED");
          }}
```

```
void DETECT(void)
{

        int CurrentNode=0,CurrentBaud=0;

        INT_CANREAD_OFF();

        NEXTIONTXT("INFO","DETECTING BAUD RATE");
        CurrentBaud=CAN.AUTO_BAUD_DETECT();
        Serial.println(CurrentBaud);
        NEXTIONTXT("BRSENSOR",(String)CurrentBaud);

        if(CurrentBaud!=0)
        {   NEXTIONTXT("INFO","Detecting Node");
            CurrentNode=CAN.NODE_DETECT();
            Serial.println(CurrentNode);
            NEXTIONTXT("NSENSOR",(String)CurrentNode);
            NEXTIONTXT("INFO","DETECTION OK");

        }

          else          if(CurrentNode==0          ||          CurrentBaud==0)
{NEXTIONTXT("NSENSOR",(String)CurrentNode); NEXTIONTXT("INFO","SENSOR NO DETECTED"); }
    }
void SDOSTRING_TO_SDONUM(String SDODATA, int *Index, int *SubIndex,long *Data)
     {
          String Dbuffer;
          String Buffer;
          Buffer=SDODATA;
          char DB[10];

           // Serial.println(Buffer);
           Dbuffer= Buffer.substring(0,4);    //Index
           Dbuffer.toCharArray(DB,6);
           //    Serial.println(Dbuffer);
           *Index =int(strtoul(DB,NULL,16));
           //Serial.println(Index,HEX);

           Dbuffer= Buffer.substring(6,8);    //Subindex
           Dbuffer.toCharArray(DB,3);
            // Serial.println(Dbuffer);
           *SubIndex =byte(strtoul(DB,NULL,16));
           //Serial.println(Subindex,HEX);

           Dbuffer= Buffer.substring(8,19);    //Index
           Dbuffer.toCharArray(DB,9);
           //      Serial.println(Dbuffer);
           *Data =(unsigned long)(strtoul(DB,NULL,16));
           // Serial.println(Data,HEX);
}


********Interrupt *********

//Interrupt to handle keyboard and menu on nextion lcd

void KEYBOARD_MENU()
```

```
{
    if(digitalRead(DOWN)==LOW)
                      { KeyEnter=0;
                        Index++;
                        if(Index<0) Index=0;
                        if(Index>MaxIndex) Index=MaxIndex;
                        NEXTION(Menu_set[Index]);
                        NEXTION(Menu_clr[Index-1]);
                        delay(2000);}


    if(digitalRead(UP)==LOW)
                    { KeyEnter=0;
                        if(Index>0)Index--;
                        if(Index<0) Index=0;
                        NEXTION(Menu_set[Index]);
                        NEXTION(Menu_clr[Index+1]);

                        delay(2000);}


    if(digitalRead(ENTER)==LOW)
                    {
                        NEXTION(Menu_Enter[Index]);
                        if(KeyEnter==0){ KeyEnter=1; LastIndex=Index;}
                        else if(KeyEnter==1&&LastIndex==Index) KeyEnter=2;
                        else if(KeyEnter==2&&LastIndex==Index) KeyEnter=0;
                        else KeyEnter=1;
                        delay(1000);

                                Serial.println(KeyEnter);
                    }

    }


//if(digitalRead(ENTER)==LOW)
//                      {
//                        NEXTION(Menu_Enter[Index]);
//                        KeyEnter=1;
//                        delay(2000);
//                      }
//Interrupt for CAN communication, INT_ON off active GETRX0 and GETRX1 at same time

      void GET_RX0(void)
                {       CAN.RXNREAD(0x00,&BRX0);
                        RXDETECTION(BRX0);
                        FRX0 =1;
                        //Serial.println("FRX0");
                }


    void GET_RX1(void)
                {     CAN.RXNREAD(0x01,&BRX1);
                    RXDETECTION(BRX1); //Flag it need to be reset it when the data i readed
                    FRX1 =1; //Serial.println("FRX1");
                 }
```

```
   void INT_CANREAD_ON(void)
        {     attachInterrupt(digitalPinToInterrupt(RX0BF), GET_RX0, FALLING);       //pin 2    = RXB0F
              attachInterrupt(digitalPinToInterrupt(RX1BF), GET_RX1, FALLING);            //pin 3=RXB1F
               }


   void INT_CANREAD_OFF(void)
           {     detachInterrupt(digitalPinToInterrupt(RX0BF));
                 detachInterrupt(digitalPinToInterrupt(RX1BF));
           }
//DATA LOGGER Interrupts

void BUFFER_RX0(void)
                 {
                        CAN.RXNREAD(0x00,&BRX0);

                        //Serial.println(BRX0.COB_ID,HEX);
                        if(IW0<=Bmax-1){//Bmax-1
                                    MsgError[IW0].N=IW0+1;
                                    MsgError[IW0].ErrorID= BRX0.COB_ID;
                                    MsgError[IW0].ErrorCode= BRX0.DATA[0] | (int(BRX0.DATA[1])<<8);
                                    MsgError[IW0].rtc=CTIME;
                                   if(IW0<Bmax-1) IW0++;
                                   else INT_Buffer_OFF();//if(IW0>Bmax-1)    IW0--;
                                    }

                         else    INT_Buffer_OFF();     //Bmax-1

                  }
void BUFFER_RX1(void)
                 {     CAN.RXNREAD(0x01,&BRX1);
                       FRX1=1;

                  }
void INT_Buffer_ON(void)
        { Serial.println("Interrupt ON");
           attachInterrupt(digitalPinToInterrupt(2), BUFFER_RX0, FALLING);       //pin 2    = RXB0F
           attachInterrupt(digitalPinToInterrupt(3), BUFFER_RX1, FALLING);            //pin 3=RXB1F
              }


void INT_Buffer_OFF(void)
           {     detachInterrupt(digitalPinToInterrupt(2));//pin 2    = RXB0F
                 detachInterrupt(digitalPinToInterrupt(3));    //pin 3=RXB1F
           }


void SensorDataFile(String SensorFile)
{     if(digitalRead(SDDETECTED)==1)
   {
   String FILE;
   char FileName[SensorFile.length()+1];

   SensorFile.toCharArray(FileName,sizeof(FileName));

   File myFile;
   String TXString_Line,Displaytxt;
```

```
        myFile = SD.open(FileName,FILE_READ);      //configuration file name

        if (myFile) {
                              Displaytxt=TXString_Line=myFile.readStringUntil('\n');
                            //    TXString_Line.toCharArray(CHDevice,14);
                                 Displaytxt.trim();      //Delete '\n' char
                              NEXTIONTXT("FILE",Displaytxt);

                              Displaytxt=TXString_Line=myFile.readStringUntil('\n');
                            // TXString_Line.toCharArray(CHBaudRate,5);
                            //    NewBaud=int(strtoul(CHBaudRate,NULL,10));

                              Displaytxt.trim(); //Delete '\n' char
                              NEXTIONTXT("BRFILE",Displaytxt);

                              Displaytxt=TXString_Line=myFile.readStringUntil('\n');
                            // TXString_Line.toCharArray(CHNode,5);
                            //    NewNode=int(strtoul(CHNode,NULL,10));

                             Displaytxt.trim(); //Delete '\n' char


                              //Displaytxt=(String)NewNode;
                              NEXTIONTXT("NFILE",Displaytxt);

                              }

            else NEXTIONTXT("INFO","FILE NOT FOUND");

          myFile.close();

        }
    else NEXTIONTXT("INFO","SD CARD NOT DETECTED");

}



********** NEXTION Functions *********

void NEXTIONTXT(String Var,String MSG)
{
  Serial2.print(Var+".txt="+"\""+MSG+"\"");
  Serial2.write(0xFF);
  Serial2.write(0xFF);
  Serial2.write(0xFF);
}

void NEXTION(String ORDER)
{                    Serial2.print(ORDER);
                      Serial2.write(0xFF);
                      Serial2.write(0xFF);
                      Serial2.write(0xFF);
  }
```

```
*************Define MCP2515 registers *******

#ifndef     MCP2515_DEFS_H
#define     MCP2515_DEFS_H

/** \name       SPI Kommandos */
/*@{*/
#define SPI_RESET           0xC0
#define     SPI_READ        0x03
#define     SPI_READ_RX             0x90
#define     SPI_WRITE               0x02
#define     SPI_WRITE_TX                0x40
#define     SPI_RTS                 0x80
#define SPI_READ_STATUS     0xA0
#define     SPI_RX_STATUS       0xB0
#define     SPI_BIT_MODIFY      0x05
/*@}*/

/** \name       Adressen der Register des MCP2515
 *
 * Die Redundanten Adressen von z.B. dem Register CANSTAT
 * (0x0E, 0x1E, 0x2E, ...) wurden dabei nicht mit aufgelistet.
 */
/*@{*/
#define RXF0SIDH    0x00
#define RXF0SIDL    0x01
#define RXF0EID8    0x02
#define RXF0EID0    0x03
#define RXF1SIDH    0x04
#define RXF1SIDL    0x05
#define RXF1EID8    0x06
#define RXF1EID0    0x07
#define RXF2SIDH    0x08
#define RXF2SIDL    0x09
#define RXF2EID8    0x0A
#define RXF2EID0    0x0B
#define BFPCTRL             0x0C
#define TXRTSCTRL   0x0D
#define CANSTAT             0x0E
#define CANCTRL             0x0F

#define RXF3SIDH    0x10
#define RXF3SIDL    0x11
#define RXF3EID8    0x12
#define RXF3EID0    0x13
#define RXF4SIDH    0x14
#define RXF4SIDL    0x15
#define RXF4EID8    0x16
#define RXF4EID0    0x17
#define RXF5SIDH    0x18
#define RXF5SIDL    0x19
#define RXF5EID8    0x1A
#define RXF5EID0    0x1B
#define TEC             0x1C
#define REC                 0x1D

#define RXM0SIDH    0x20
```

```
#define RXM0SIDL   0x21
#define RXM0EID8   0x22
#define RXM0EID0   0x23
#define RXM1SIDH   0x24
#define RXM1SIDL   0x25
#define RXM1EID8   0x26
#define RXM1EID0   0x27
#define CNF3       0x28
#define CNF2       0x29
#define CNF1       0x2A
#define CANINTE        0x2B
#define CANINTF        0x2C
#define EFLG       0x2D

#define TXB_0              0x30    //Define header for TXB0 message its the same as TXB0CTRL
#define TXB0CTRL   0x30
#define TXB0SIDH   0x31
#define TXB0SIDL   0x32
#define TXB0EID8   0x33
#define TXB0EID0   0x34
#define TXB0DLC        0x35
#define TXB0D0         0x36
#define TXB0D1         0x37
#define TXB0D2         0x38
#define TXB0D3         0x39
#define TXB0D4         0x3A
#define TXB0D5         0x3B
#define TXB0D6         0x3C
#define TXB0D7         0x3D

#define TXB_1              0x40     //Define header for TXB1 message its the same as TXB1CTRL
#define TXB1CTRL   0x40
#define TXB1SIDH   0x41
#define TXB1SIDL   0x42
#define TXB1EID8   0x43
#define TXB1EID0   0x44
#define TXB1DLC        0x45
#define TXB1D0         0x46
#define TXB1D1         0x47
#define TXB1D2         0x48
#define TXB1D3         0x49
#define TXB1D4         0x4A
#define TXB1D5         0x4B
#define TXB1D6         0x4C
#define TXB1D7         0x4D

#define TXB_2              0x50     //Define header for TXB2 its the same addres than TXB2CTRL
#define TXB2CTRL   0x50
#define TXB2SIDH   0x51
#define TXB2SIDL   0x52
#define TXB2EID8   0x53
#define TXB2EID0   0x54
#define TXB2DLC        0x55
#define TXB2D0         0x56
#define TXB2D1         0x57
#define TXB2D2         0x58
#define TXB2D3         0x59
```

```
#define TXB2D4          0x5A
#define TXB2D5          0x5B
#define TXB2D6          0x5C
#define TXB2D7          0x5D

#define RXB0CTRL    0x60
#define RXB0SIDH    0x61
#define RXB0SIDL    0x62
#define RXB0EID8    0x63
#define RXB0EID0    0x64
#define RXB0DLC         0x65
#define RXB0D0          0x66
#define RXB0D1          0x67
#define RXB0D2          0x68
#define RXB0D3          0x69
#define RXB0D4          0x6A
#define RXB0D5          0x6B
#define RXB0D6          0x6C
#define RXB0D7          0x6D

#define RXB1CTRL    0x70
#define RXB1SIDH    0x71
#define RXB1SIDL    0x72
#define RXB1EID8    0x73
#define RXB1EID0    0x74
#define RXB1DLC         0x75
#define RXB1D0          0x76
#define RXB1D1          0x77
#define RXB1D2          0x78
#define RXB1D3          0x79
#define RXB1D4          0x7A
#define RXB1D5          0x7B
#define RXB1D6          0x7C
#define RXB1D7          0x7D
/*@}*/

/** \name       Bitdefinition der verschiedenen Register */
/*@{*/

/** \brief Bitdefinition von BFPCTRL */
#define B1BFS       5
#define B0BFS       4
#define B1BFE       3
#define B0BFE       2
#define B1BFM       1
#define B0BFM       0

/** \brief Bitdefinition von TXRTSCTRL */
#define B2RTS       5
#define B1RTS       4
#define B0RTS       3
#define B2RTSM          2
#define B1RTSM          1
#define B0RTSM          0

/** \brief Bitdefinition von CANSTAT */
#define OPMOD2          7
```

```
#define OPMOD1          6
#define OPMOD0          5
#define ICOD2       3
#define ICOD1       2
#define ICOD0       1

/** \brief Bitdefinition von CANCTRL */
#define REQOP2          7
#define REQOP1          6
#define REQOP0          5
#define ABAT        4
#define CLKEN       2
#define CLKPRE1         1
#define CLKPRE0         0

/** \brief Bitdefinition von CNF3 */
#define WAKFIL          6
#define PHSEG22         2
#define PHSEG21         1
#define PHSEG20         0

/** \brief Bitdefinition von CNF2 */
#define BTLMODE         7
#define SAM             6
#define PHSEG12         5
#define PHSEG11         4
#define PHSEG10         3
#define PHSEG2          2
#define PHSEG1          1
#define PHSEG0          0

/** \brief Bitdefinition von CNF1 */
#define SJW1        7
#define SJW0        6
#define BRP5        5
#define BRP4        4
#define BRP3        3
#define BRP2        2
#define BRP1        1
#define BRP0        0

/** \brief Bitdefinition von CANINTE */
#define MERRE       7
#define WAKIE       6
#define ERRIE       5
#define TX2IE       4
#define TX1IE       3
#define TX0IE       2
#define RX1IE       1
#define RX0IE       0

/** \brief Bitdefinition von CANINTF */
#define MERRF       7
#define WAKIF       6
#define ERRIF       5
#define TX2IF       4
#define TX1IF       3
```

```
#define TX0IF		2
#define RX1IF		1
#define RX0IF		0

/** \brief Bitdefinition von EFLG */
#define RX1OVR		7
#define RX0OVR		6
#define TXB0		5
#define TXEP		4
#define RXEP		3
#define TXWAR		2
#define RXWAR		1
#define EWARN		0

/** \brief Bitdefinition von TXBnCTRL (n = 0, 1, 2) */
#define ABTF		6
#define MLOA		5
#define TXERR		4
#define TXREQ		3
#define TXP1		1
#define TXP0		0

/** \brief Bitdefinition von RXB0CTRL */
#define RXM1		6
#define RXM0		5
#define RXRTR		3
#define BUKT		2
#define BUKT1		1
#define FILHIT0		0

/** \brief Bitdefinition von TXBnSIDL (n = 0, 1) */
#define	EXIDE		3

/**
 * \brief	Bitdefinition von RXB1CTRL
 * \see		RXM1, RXM0, RXRTR und FILHIT0 sind schon fuer RXB0CTRL definiert
 */
#define FILHIT2		2
#define FILHIT1		1

/** \brief Bitdefinition von RXBnSIDL (n = 0, 1) */
#define	SRR		4
#define	IDE		3

/**
 * \brief	Bitdefinition von RXBnDLC (n = 0, 1)
 * \see		TXBnDLC	(gleiche Bits)
 */
#define	RTR		6
#define	DLC3		3
#define	DLC2		2
#define	DLC1		1
#define DLC0		0

/*@}*/
#endif	// MCP2515_DEFS_H
```