Sistema Naval de Adquisición de Datos

Virginia Garrido San Martín
Máster Universitario en Ingeniería de Telecomunicación UOC-URL
Electrónica

Carlos Monzo Sánchez
Aleix López Antón

Junio 2017

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Sistema Naval de Adquisición de Datos*
Nombre del autor:	*Virginia Garrido San Martín*
Nombre del consultor/a:	*Aleix López Antón*
Nombre del PRA:	*Carlos Monzo Sánchez*
Fecha de entrega (mm/aaaa):	06/2017
Titulación:	*Máster Universitario en Ingeniería de Telecomunicación UOC-URL*
Área del Trabajo Final:	*Electrónica*
Idioma del trabajo:	*Castellano*
Palabras clave	*Electrónica, naval, Intel Galileo*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

El presente TFM se centra en el estudio de una alternativa que permita modernizar el hardware de las PYMES navales debido a que actualmente nos encontramos con electrónica desfasada, que no alcanza su potencial y por tanto limitada en funcionalidad y seguridad.

Para resolver este problema se va a proceder al diseño de un nuevo producto (utilizando SoC inicialmente pensados para la educación) que pueda sustituir las tarjetas de adquisición de datos actuales.

Durante la fase de implementación de este proyecto hemos conseguido realizar un equipo (como boceto inicial) que permite la sustitución de los elementos actuales, así como servir como alternativa en caso de actualizar el protocolo de comunicación (como en el sector industrial está sucediendo).

Con esto se consigue iniciar una vía de desarrollo aplicada al sector naval que puedan llevar a cabo PYMES o pequeños equipos de desarrollos con una

inversión mínima que permita mantener los sistemas de monitorización de señales a la última en tecnología en todo momento.

Abstract (in English, 250 words or less):

This Master Thesis focuses on the study of an alternative that allows to modernize the hardware of the naval SMEs because now they have outdated electronics, that does not reach its potential and therefore are limited in functionality and security.

To solve this problem, we will proceed to design a new product (using SoC initially intended for education) that can replace the current data acquisition cards.

At the implementation phase of this project, we have could create an equipment (as initial design) that allows the replacement of the current elements, as well serve as an alternative in case of updating the communication protocol (as in the industrial sector is happening).

With this is possible to start a development path applied to the naval sector that can carry out SMEs or small development teams with a minimum investment that allows to keep signal monitoring systems to the latest in technology always.

Índice

Lista de figuras

Lista de tablas

1. Introducción

1.1. Contexto y justificación del Trabajo

El auge del sector naval que se está viviendo en los últimos años junto con los avances en la tecnología ha implicado que los armadores soliciten mejores y más complejos sistemas de IAS (Integrated Alarm System). Estos sistemas se encargan básicamente en mantener una monitorización constante de sensores tanto analógicos como digitales, así como de ejercer el control sobre equipos externos tales como bombas y válvulas, enviando esta información por un cableado, habitualmente CAN Bus, que recorre el interior de los barcos.

Sin embargo, a nivel de hardware nos encontramos con equipos desfasados, que no alcanzan todo su potencial debido a la gran inversión que pone para una pequeña/mediante empresa readaptar constantemente las placas base de todos sus dispositivos con las consiguientes certificaciones navales que esto implica.

Es por este motivo que el presente TFM se centra en la búsqueda de una alternativa basada en hardware libre que permita modernizar el hardware de la PYMES navales cumpliendo con los requisitos que se aplican a este tipo de monitorización, así como poder adaptarse fácilmente al protocolo Ethernet ya que es el futuro sustituto del protocolo CAN Bus.

1.2. Objetivos del Trabajo

El objetivo principal de este proyecto es diseñar un sistema de adquisición de datos que pueda ser utilizado en el ámbito naval utilizando sistemas de hardware libre para estudiar su viabilidad frente a la electrónica analógica discreta que están utilizando los sistemas de monitorización para buques.

1.3. Enfoque y método seguido

La estrategia seleccionada para llevar a cabo este trabajo es el desarrollo de un nuevo producto que sustituya las tarjetas de adquisición de datos actuales. Para poder realizar este desarrollo previamente se tiene que realizar un estudio de los

SoC disponibles en el mercado, así como un análisis teórico de los principios de los protocolos CAN Bus y Ethernet.

Se van a realizar dos diseños distintos, uno para cada protocolo debido a que actualmente se utiliza el protocolo CAN Bus, pero en los próximos años se prevé una sustitución de éste por el protocolo Ethernet.

1.4. Planificación del Trabajo

1.4.1. Recursos Necesarios

Para la realización de este Trabajo Final de Máster son necesarios los siguientes recursos:

- 2 SoC Intel Galileo
- 2 Shield CAN Bus
- Cables Macho-Macho
- Resistencias
- Diodo 1N4148
- Cable Ethernet
- Router
- 2 Cables USB a USB tipo C
- Ordenador Personal (en este caso se va a utilizar Windows, pero son válidos otros sistemas operativos).
- Sensor LM35
- Sensor DHT11

A nivel de Software se van a utilizar los siguientes programas y aplicaciones:

- Arduino IDE
- Eclipse
- Fritzing
- Microsoft Edge
- NotePad++ (versión de 32 bits)
- Gimp

1.4.2. Tareas a Realizar

Las tareas a realizar tienen cierta relación con los entregables, es decir, los productos obtenidos, que explicaremos brevemente en el capítulo 1.5, pero a grandes rasgos podríamos dividirlas en las siguientes:

- Definir el ámbito del proyecto.
- Realización de un estado del arte.
- Realizar un estudio teórico de las comunicaciones.
- Estudiar los distintos dispositivos del mercado.
- Especificar los recursos hardware necesarios.
- Especificar los recursos software necesarios.
- Adquirir el material necesario.
- Diseñar los circuitos a conectar.
- Instalación y puesta a punto del software a utilizar.
- Diseño del producto final.
- Programación de la lectura de los sensores.
- Programación del envío y recepción de datos por CAN.
- Programación del servidor web
- Implementación del producto final.
- Testeo de la Implementación.
- Corrección de los errores

1.4.3. Planificación Temporal

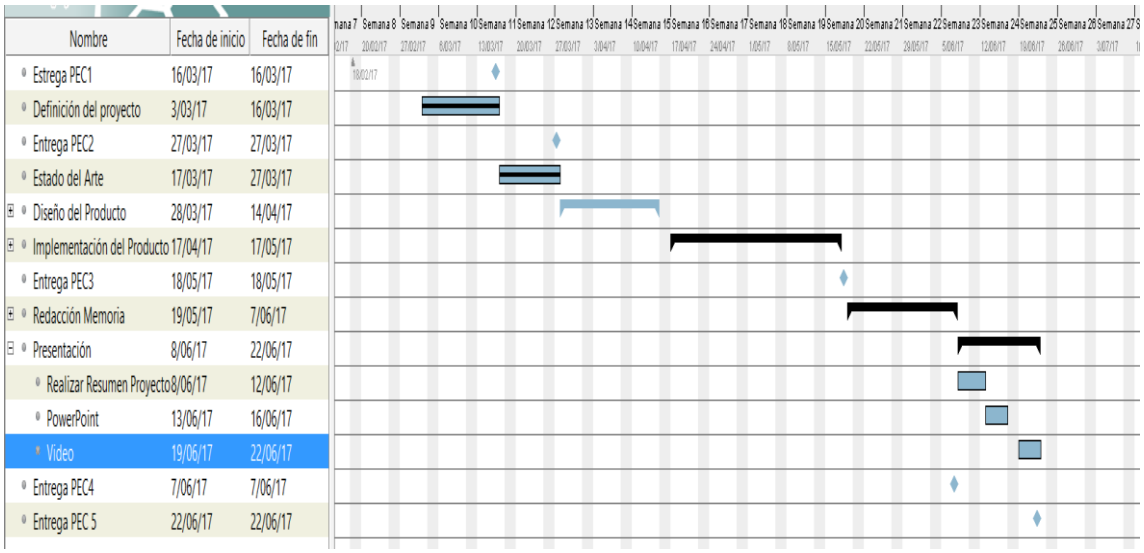

Ilustración 1: Planificación Gantt. Fuente Propia

1.5. Breve sumario de productos obtenidos

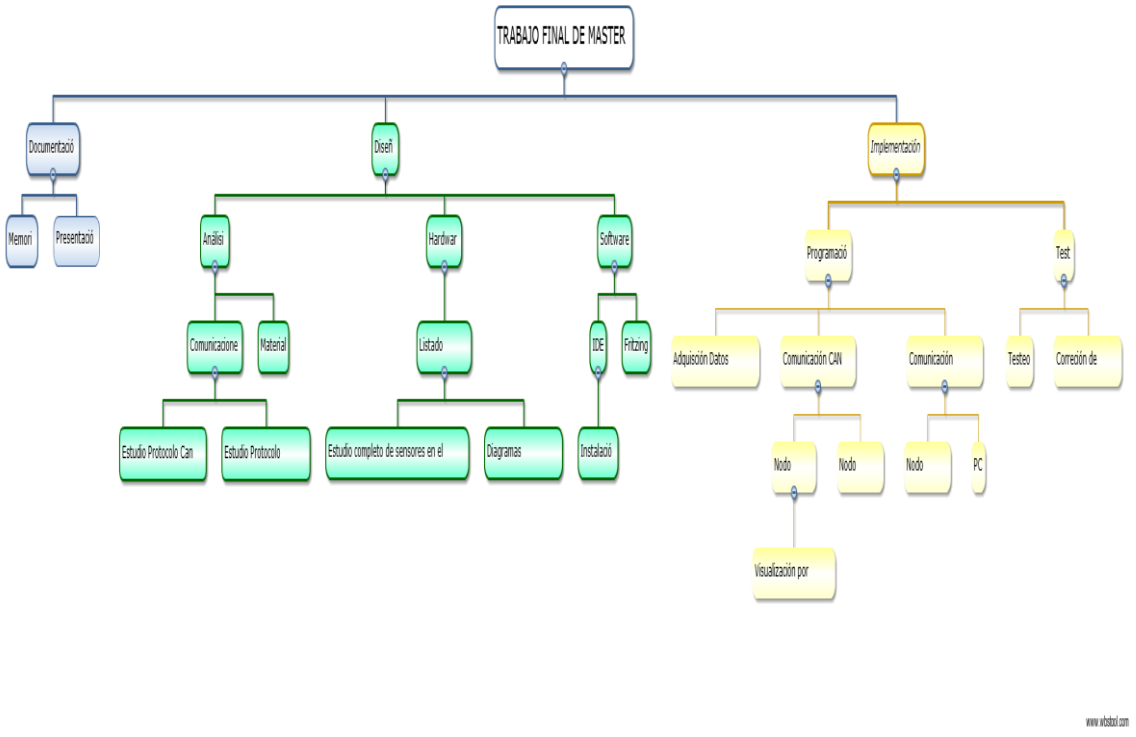

Ilustración 2: Gráfico de Entregables. Fuente Propia

Tal y como se ve en la Ilustración 2, este TFM se divide en tres grandes entregables (documentación, diseño e implementación). El entregable de documentación se divide en:

- Memoria
- Presentación

El entregable de diseño se divide en:

- Análisis. Consiste en realizar los estudios teóricos previos al diseño.
- Hardware. Incluye listar el hardware necesario, adquirirlo y realizar los diagramas de conexionado de los circuitos a implementar.
- Software. Formado por la búsqueda e instalación del software necesario.

En el entregable de implementación dejamos las partes de programar las lecturas y envíos de datos, así como las pruebas y correcciones de errores, en caso de observar alguno.

1.6. Breve descripción de los otros capítulos de la memoria

Esta memoria comienza con el estado del arte del proyecto donde se explica brevemente la historia de los sistemas de adquisición de datos y el estado actual de estos sistemas aplicados al sector naval.

Posteriormente, se explica que son los SoC y las ventajas que traen los SoC de tipo hardware libre. Tras esto se realiza un estudio de las placas disponibles en el mercado.

En los capítulos 4 y 5 se realiza una explicación teórica de los fundamentos básicos del funcionamiento de los protocolos de comunicación CAN Bus y Ethernet a nivel de capa física y de enlace.

Tras esto, estamos preparados para realizar el diseño de la solución y su posterior implementación en el capítulo 7.

Finalizamos este proyecto con el capítulo 9 y las conclusiones que se han obtenido tras el desarrollo de éste.

2. Estado del arte

En este capítulo voy a exponer el conocimiento adquirido tras la realización de la búsqueda bibliográfica que me ha permitido encauzar este trabajo final de máster en la dirección correcta.

Para esto, comenzaremos explicando brevemente la historia de los sistemas de adquisición de datos, para tener claro de dónde partimos, para centrarnos en la situación actual de estos sistemas ya centrados en el sector naval para así definir hacia dónde vamos.

Posteriormente compararemos este trabajo final de máster con otros proyectos similares para obtener las diferencias (qué puntos son novedosos en este trabajo final de master) así como las similitudes (qué puntos podemos utilizar para realizar con éxito la presente tarea).

Daremos por finalizado este capítulo con una breve conclusión donde resumiremos que beneficios y novedades puede aportar este proyecto a la situación de los sistemas de adquisición de datos navales.

2.1. Historia de los sistemas de adquisición de datos

En la actualidad, los avances de la electrónica y la miniaturización de ésta, han motivado un auge en la automatización en el sector industrial, en el sector transporte, en el naval, que es el que nos incumbe en este proyecto, y hasta en el propio hogar. Pero toda automatización parte de un sistema de adquisición de datos.

Este sistema consiste en un elemento que toma muestras de una señal externa a él, normalmente proveniente de un sensor, las convierte en tensiones eléctricas, es decir, se acondiciona la señal y posteriormente la digitaliza para permitir trabajar con ella[1].

Estos sistemas existen teóricamente desde 1927, año en que Nyquist definió su teorema del muestreo[1] que permite reconstruir señales en banda base a partir

[1] La frecuencia de muestreo mínima requerida para realizar una digitalización de calidad, debe ser igual al doble de la frecuencia de la señal analógica que se pretenda digitalizar.[2]

de sus muestras[2], sin embargo, no pudo aplicarse hasta la invención de un circuito electrónico conversor de analógico a digital.

Es difícil decir con claridad cuando se fabricó por primera vez un sistema conversor de datos ya que los primeros que existieron no eran electrónicos, si no que eran hidráulicos [3, p. 3]. Uno de los primeros conversores hidráulicos de los cuales tenemos constancia data del siglo XVIII[4].

Este sistema hidráulico usaba depósitos que se mantenían a una profundidad constante, lo que equivale una referencia de potencial si lo vemos desde un punto de vista electrónico, por medio de unas canalizaciones por el cual el agua casi no circula. El agua que sale del depósito se controlaba mediante unas boquillas de estados binarios sumergidas 96 mm [4, pp. 165-167] por debajo de la superficie del agua. El tamaño de estas boquillas corresponde a múltiplos binarios de una unidad básica de medida llamada 1 lüle lo cual equivale a unos 36 litros por minuto. Si nos fijamos, vemos que está funcionando como un conversor digital/analógico de 8 bits.

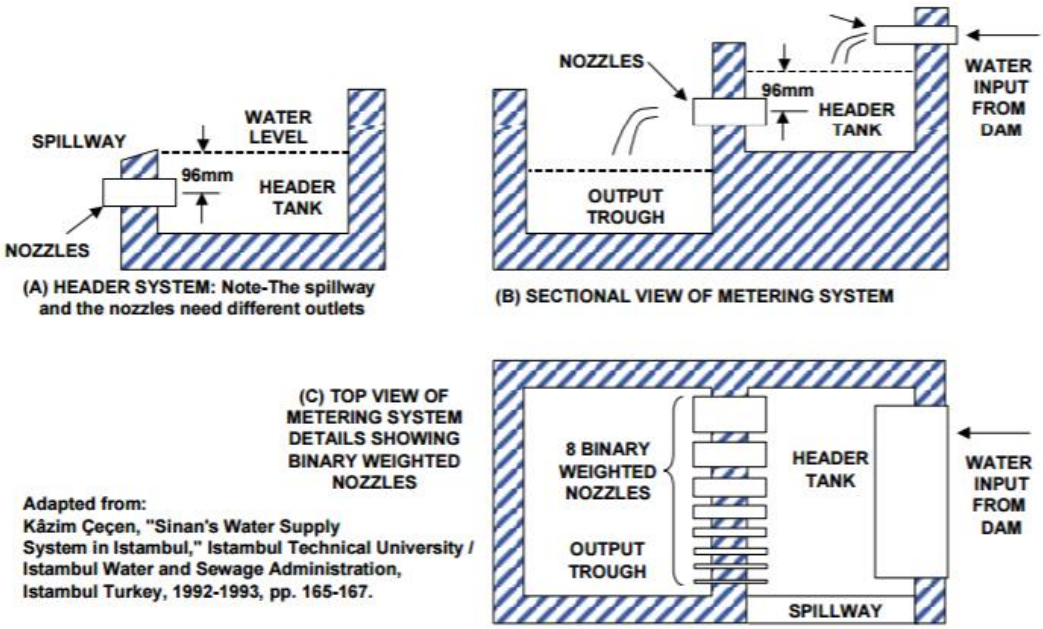

ILUSTRACIÓN 3: SISTEMA HIDRÁULICO DE CONVERSIÓN ANALÓGICO-DIGITAL. OBTENIDO DE [3]

Estos sistemas se mantuvieron hasta el desarrollo de los conversores de datos electrónicos para sus aplicaciones en comunicaciones. El telégrafo nos llevó a la invención del teléfono y este a la formación del sistema Bell[5].

La proliferación del telégrafo y posteriormente del teléfono junto con la demanda de más capacidad acarreó la necesidad de multiplexar más de un canal en un único par de cobre[6]. Aunque la multiplexación por división de tiempo (TDM) era la más popular, la multiplexación por frecuencia era la más útil y usada. Sin embargo, no fue hasta la invención de la modulación por pulsos (PCM) que surgieron los conversores como los entendemos actualmente.

La modulación por pulsos fue inventada en 1921 por Paul Rainey[7]. Su patente describe un método para transmitir los datos codificados de un telégrafo utilizando 5 bits. Como hemos comentado anteriormente, Harry Nyquist estudiando estas señales definiría posteriormente su teorema que definiría hasta la actualidad la modulación PCM.

Tendremos que esperar a 1937 a una patente de Alec Reeves [8] para encontrar los primeros conversores completamente electrónicos. Este sistema básicamente usaba un muestreo por pulsos para tomar una muestra de la señal analógica, la cual pasaba por un flip-flop de tipo RS que activaba una rampa de voltaje. La rampa se comparaba con la entrada y cuando eran igual se generaba un pulso que reseteaba el flip-flop. La salida de este biestable es un pulso cuyo ancho de banda es proporcional a la señal analógica muestreada. Este pulso posteriormente era convertido a una señal digital utilizando un simple contador.

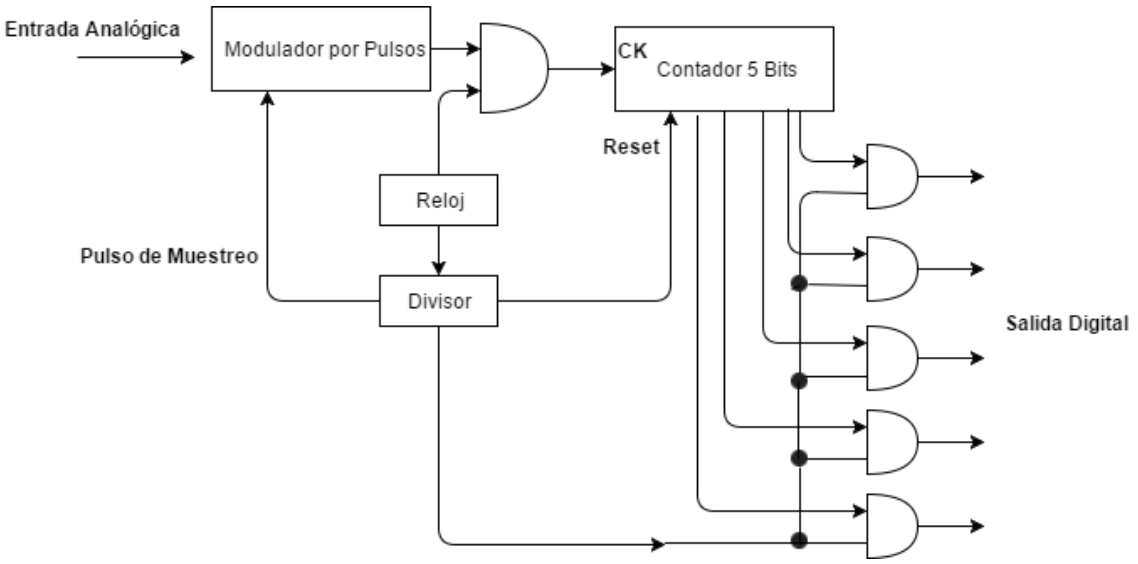

ILUSTRACIÓN 4: ADAPTACIÓN DEL CIRCUITO DE LA PATENTE DE REEVES. FUENTE PROPIA.

La patente de Reeves cubre todos los procesos de adquisición de datos: muestreo, cuantificación y codificación de las muestras digitalizadas para su posterior tratamiento.

Con el invento de los transistores de silicio en 1954 por Gordon Teal se abrió las puertas al desarrollo de la electrónica moderna, siendo uno de los primeros inventos en aparecer algo tan habitual a día de hoy como los circuitos integrados en 1958 [9].

A raíz de la aparición de los circuitos integrados, estos sistemas caros, espaciosos y que disipaban potencia excesiva que solo estaban pensados para un ámbito de telefonía o militar comienzan a abrirse a aplicaciones industriales, científicas y educativas. En concreto, comienzan a aparecer las llamadas tarjetas de adquisición de datos.

Las tarjetas de adquisición de datos, también conocidas como dispositivos DAQ [1] son un interfaz entre las señales analógicas producidas por una serie de sensores y un PC. Integra por tanto en un único dispositivo el sistema de acondicionado de señal, un circuito conversor analógico/digital (incluyendo por tanto su circuito de muestreo y cuantificación) y un bus de salidas digitales para facilitar el conexionado al pc.

La gran ventaja de estas tarjetas frente a los circuitos por separado como teníamos antes es que no hay que preocuparse por realizar un sistema de acondicionado de señal específico para cada sensor [10] puesto que ya vienen implementados en la misma tarjeta. A su vez, están aislados protegiendo así al circuito y haciéndolo más duradero.

Existen diferentes tipos de dispositivos DAQ en función de la cantidad de señales analógicas a tratar. Un ejemplo de uso general podría ser el que tenemos en la siguiente imagen:

ILUSTRACIÓN 5: DISPOSITIVO DAQ NI920. OBTENIDO DE [11]

El dispositivo NI9201 [11], es capaz de monitorizar hasta 8 canales analógicos a la vez y su precio ronda los 500€. Por si solas los dispositivos DAQ no son capaces de funcionar, así que a este precio habría que añadir un ordenador que realice el tratamiento de los datos y un módulo de comunicación en caso de disponer de varios nodos de dispositivos DAQ.

Debido a la tendencia a la miniaturización de la electrónica, junto con el elevado coste que supone para algunos sectores las tarjetas de adquisición de datos, se comenzaron a utilizar los procesadores, creados inicialmente para teléfonos móviles, que integran en un único chip todos los componentes del sistema. Llegaron así los dispositivos SOC.

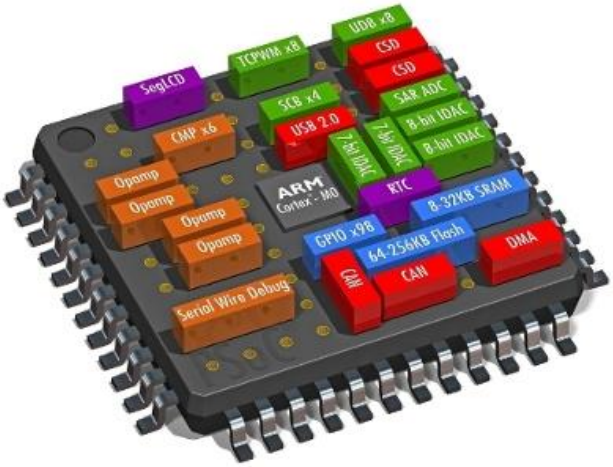

ILUSTRACIÓN 6:IMAGEN REPRESENTATIVA DE UN SOC. OBTENIDA DE [12]

El diseño de estos sistemas está basado en circuitos de señal mixta e incluso algunos de ellos presentan sistemas de radiofrecuencia como Bluetooth o Wifi.[13] Por tanto, un SoC estándar está formado por:

- Un microprocesador o microcontrolador.
- Módulos de memoria ROM, RAM, EEPROM y Flash.
- PLL para generar las diferentes señales de reloj internas.
- Controladores de comunicación externa como UART, SPI o un USB.
- Controladores de interfaces digitales.
- Controladores de interfaces analógicos incluyendo ADC y DAC.

Esto presenta una ventaja frente a las tarjetas de adquisición de datos: no solo tenemos el sistema de adquisición si no también la capacidad de procesarlos sin necesidad de un pc.

Trataremos en profundidad los diferentes dispositivos SoC más populares del mercado incluyendo una comparativa en capítulos siguientes.

2.2. Situación actual de los sistemas de adquisición del sector naval

Desde que en el milenio III antes de Cristo, los fenicios comenzaran a desarrollar la construcción naval, tal y como hoy la conocemos, hasta la actualidad los barcos se han ido aprovechando de los avances tecnológicos para mejorar sus

sistemas que faciliten ya sea la navegación, como los sistemas GPS y de master clock, el sistema de flotación, con los sistemas antirolling y de auto-ballast, pasando por la seguridad con los sistemas de Integrated Alarm System (IAS).

Los sistemas de IAS consisten básicamente en un sistema de adquisición de datos analógicos y/o digitales, normalmente señales de alarmas o que dadas ciertas condiciones se consideren alarmas. También se reciben señales para el control de equipos como válvulas, bombas y ventiladores.

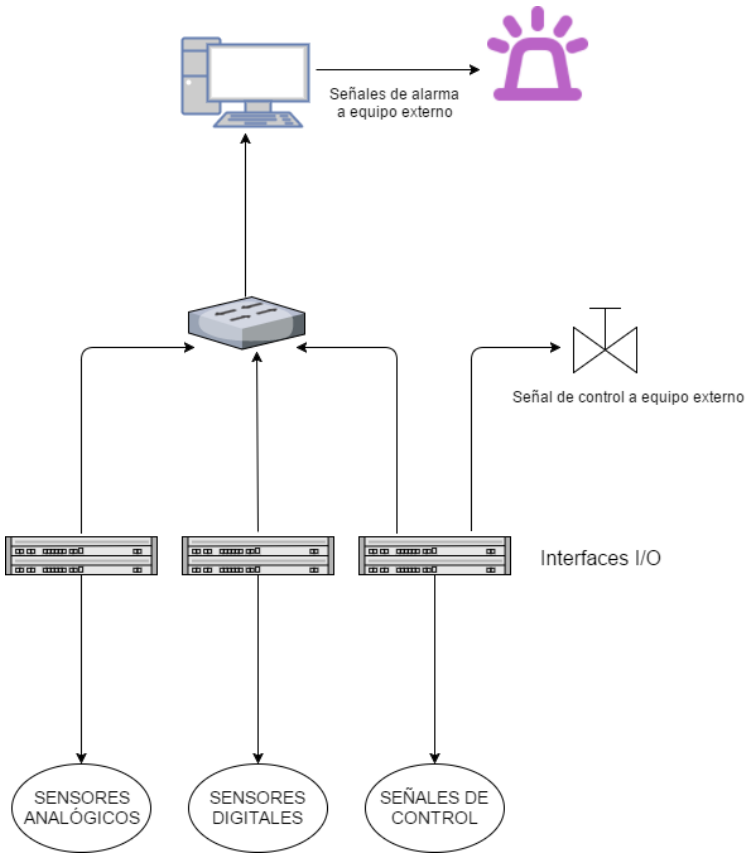

ILUSTRACIÓN 7: DIAGRAMA GENERAL DE UN SISTEMA IAS. FUENTE PROPIA.

Para poder explicar con detenimiento el estado actual de estos sistemas, lo vamos a realizar mediante el estudio de ellos aplicados a 4 de las mayores empresas de diseño de éstos.

2.2.4. Høglund

Høglund [14] es una empresa noruega de automatización del sector naval. Su principal producto son los IAS. Éste se basa en los dispositivos ABB S800 [15] como tarjeta de adquisición de datos. Posteriormente los datos digitalizados

pasan a un dispositivo de controlador de procesos ABB AC800M tras lo cual se envía a un ordenador para que finalice el procesamiento de los datos.

Los módulos ABB S800 (con precios de segunda mano de unos 350€) se comunican entre ellos mediante profibus con lo cual necesitan conectar los dispositivos lo más cercano posible a los equipos de los que leer las señales. El módulo ABB AC800M se comunica a su vez vía LAN con los ordenadores (denominadas estaciones en este ámbito).

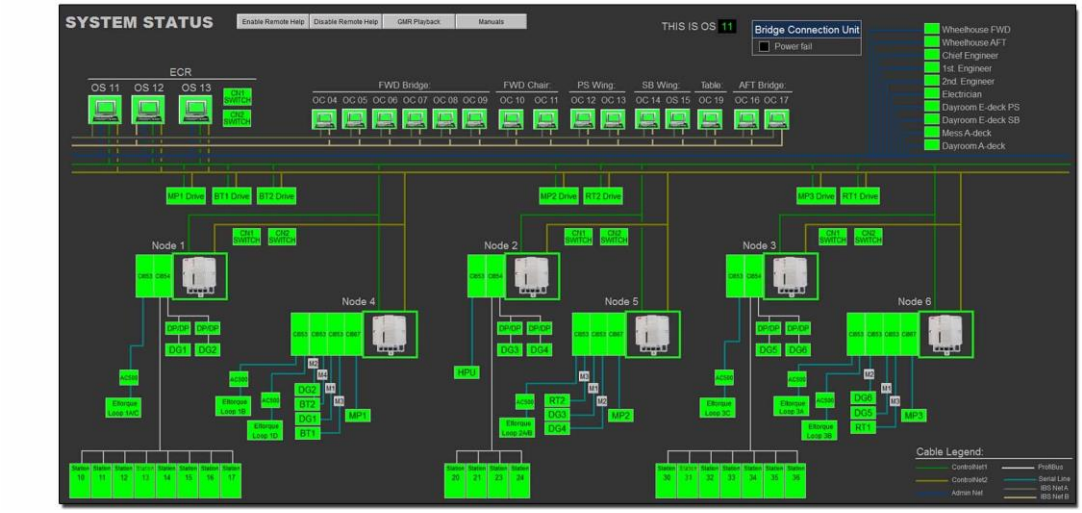

ILUSTRACIÓN 8: DISTRIBUCIÓN DE UN SISTEMA DE HØGLUND [14]

2.2.5. Ingeteam

Ingeteam [16] es una empresa internacional fundada en 1972, con sede en el País Vasco, especializada en electrónica de potencia y de control. A pesar de no ser su producto insignia, sus sistemas de IAS son muy reputados.

En este caso el sistema está formado por PLC de terceros (Scheiner, etc.) y ordenadores siguiendo el esquema estándar. Los PLCs se comunican entre ellos utilizando una red de redundancia de CAN Bus, con lo cual pueden situarlos en cualquier lugar del barco. Al igual que Hoglund, comunican los nodos con los ordenadores mediante LAN.

No hay información al público en general sobre los precios de sus productos, pero podemos hacernos una idea teniendo en cuenta que los COST (Commercial Off-The-Shelf) PLCs que no presentan una certificación que les permita instalarse en buques rondan los 300€.

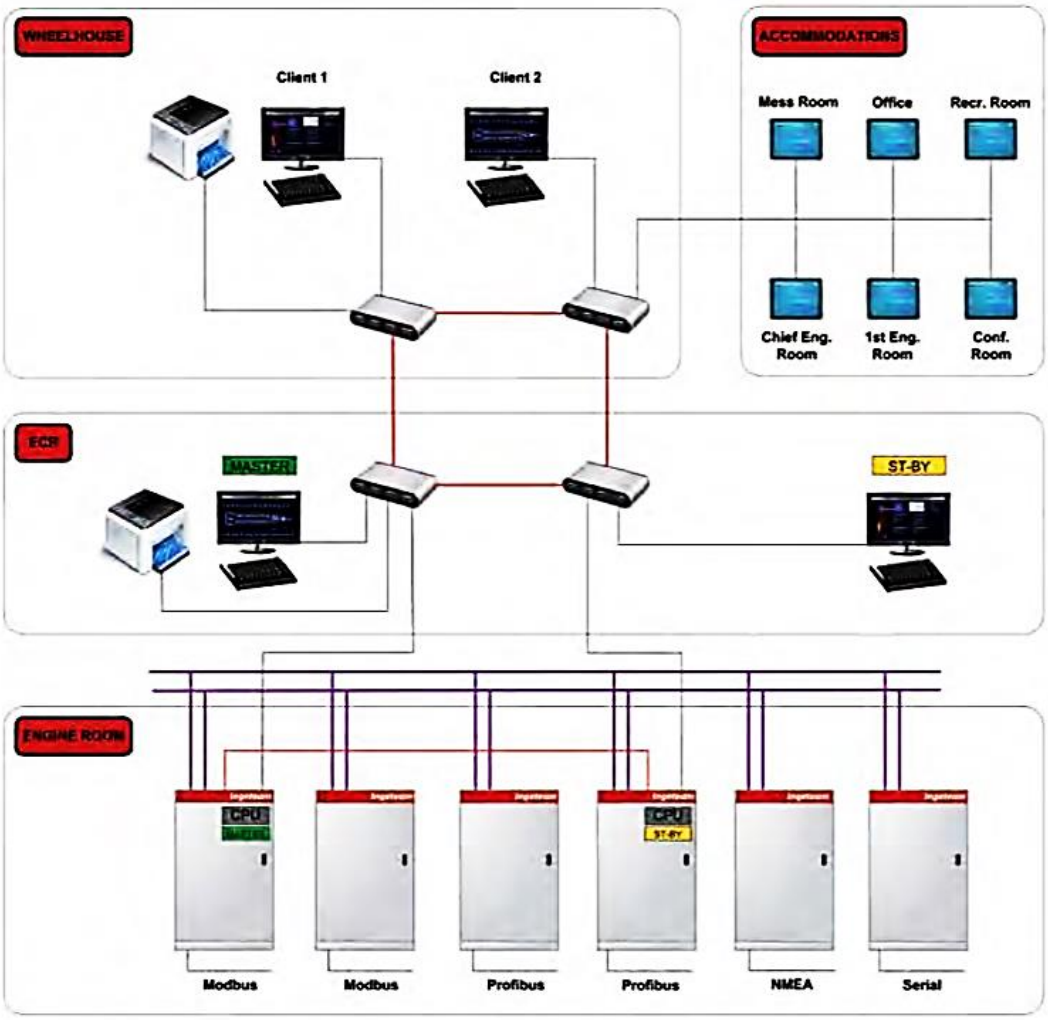

ILUSTRACIÓN 9: DISTRIBUCIÓN DE UN SISTEMA DE INGETEAM. OBTENIDO DE [16]

2.2.6. Sedni

Sedni [17] es una empresa española fundada en 1987 con sede en Alicante. Su producto estrella es Diamar, nombre con el que denominan el IAS.

Esta empresa fabrica todos los dispositivos del sistema de adquisición de datos. Tenemos, así, unas tarjetas de adquisición de datos de entrada/salida (en función del tipo de tarjeta) que se comunican entre ellas vía triple-can bus. A su vez se comunican directamente también con algunos de sus computadores utilizando la misma red Can Bus. Estos computadores que reciben la señal can bus, la duplican por LAN al resto de ordenadores y dispositivos.

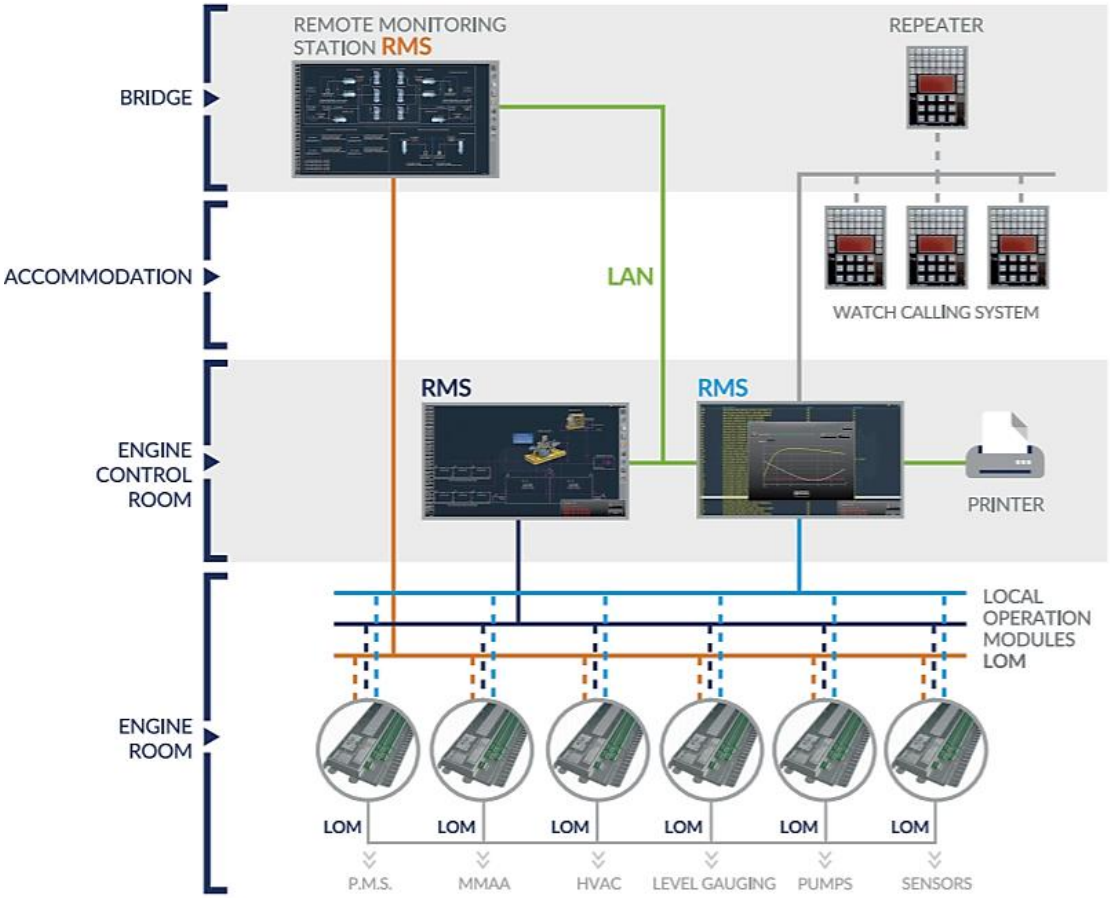

ILUSTRACIÓN 10: DISTRIBUCIÓN DEL SISTEMA DE SEDNI. OBTENIDA DE [17]

Los precios de sus LOM (las tarjetas de adquisición de datos) se encuentran entre 450 y 600€ según las características de éstas.

2.2.7. Wärtsilä

Wärsilä [18] es una empresa noruega fundada en 1979. En su momento se dedicaba a la automatización industrial pero actualmente y desde 1983 se dedica única y exclusivamente a la automatización naval.

Esta empresa tiene el honor de haber diseñado el primer sistema de IAS para el mercado naval.

Al igual que SEDNI, fabrica sus propias tarjetas de adquisición de datos. La diferencia con los sistemas de las empresas anteriores es que en este caso se comunican por CanBus o línea serie con los equipos a monitorizar y la salida la

envían a un dispositivo que se encuentra formando un anillo de lan por toda la superficie del barco.

Siguen manteniendo el procesado de las señales utilizando un ordenador que se conecta a la red LAN.

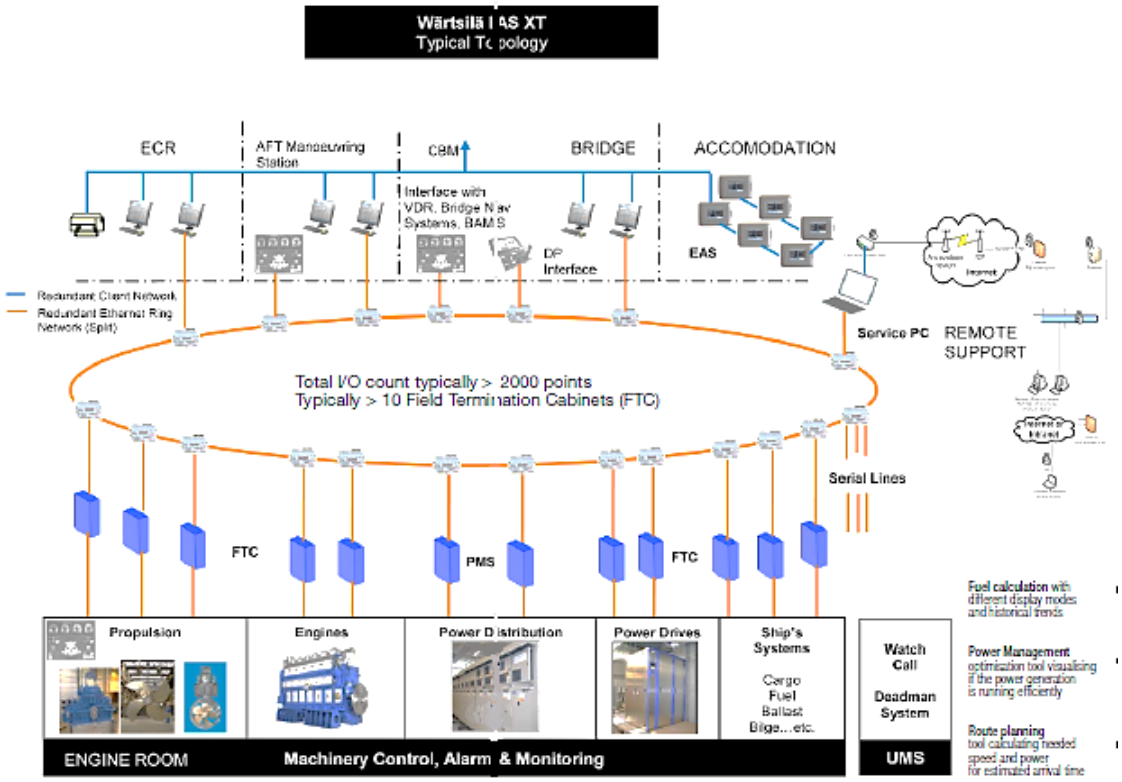

ILUSTRACIÓN 11: DISTRIBUCIÓN DE UN SISTEMA WÄRTSILÄ ADAPTADA DE [18]

Tras este breve análisis vemos que actualmente los sistemas de adquisición de datos se siguen basando en distintas versiones de tarjetas de adquisición de datos separando así la adquisición del procesado, en lugar de unificarlo con un SoC.

2.3. Proyectos similares

Nos hemos encontrado con un vacío de proyectos realizados hasta la fecha sobre esta temática, pero vamos a explicar los tres proyectos más semejantes que hemos encontrado.

En [19] encontramos un proyecto que se basa en utilizar una Arduino para leer vía Wifi los datos de un sensor MARG[2] situado en un bote de remos. En este caso vemos que se ha implementado un sistema de monitorización de la posición para un bote sencillo utilizando SoC.

Aunque se plantee la instalación en un bote de remos, si nos fijamos en la monitorización y procesado de los datos se realiza fuera de éste, instalando allí solamente el sensor y un dispositivo XBee para el envío de los datos de manera inalámbrica.

De este proyecto tenemos la similitud del sector y la utilización de un SoC, pero se difiere en el tipo de sensores a utilizar y protocolo de comunicación ya que en nuestro caso se trata de CANBus para que sea la más similar posible a los sistemas de IAS que hemos tratado en la página **¡Error! Marcador no definido.**.

En [20] encontramos la creación de una red CAN Bus mediante Arduino para monitorizar los datos que un vehículo utilizando un cableado OBD2. Para poder realizar este proyecto, primero realiza un estudio del protocolo CAN tanto teórico como práctico, ya que realiza una comunicación entre 3 Arduino para demostrar su eficacia. Posteriormente, como hemos comentado, utiliza una única Arduino como máster de una comunicación con un vehículo, siendo necesario que el operador le indique que valor leer puesto que no realiza una monitorización continua.

Encontramos en este caso más similitudes que en el proyecto [19] puesto que realiza al igual que nosotros una comunicación mediante CAN Bus así como un estudio teórico previo de este protocolo. La diferencia la encontramos en el hecho de que no vamos a interactuar con elementos externos, dejando que tanto el maestro como el esclavo sean dispositivos SoC. Debido a esto, necesitamos que nuestros dispositivos además de interactuar entre ellos con el protocolo CAN Bus, tienen que realizar la adquisición de datos (muestreo, cuantificación y digitalización) de los sensores analógicos y digitales que se conectarán al nodo esclavo.

Finalmente, el último proyecto encontrado [21] consiste en el diseño de un sistema de control de temperatura para una jarra de agua. Este sistema se basa

[2] Magnetic, Angular Rate and Gravitational.

en un SoC (concretamente Arduino) que realiza una adquisición de datos desde un sensor y utiliza como máster un PC que procesa estos datos y los muestra por pantalla utilizando LabView. En este caso, al igual que en el proyecto [19] la comunicación se realiza de modo inalámbrico. Posteriormente tras procesar los datos y en caso de ser necesario actuar, el PC manda una orden a la tarjeta de adquisición de datos para que actúe sobre un relé.

La primera parte de éste proyecto es igual que el nuestro, utilizar un SoC como tarjeta de adquisición de datos de unos sensores, sin embargo, a partir de aquí nos desligamos completamente ya que en este proyecto no vamos a utilizar un PC como procesador si no otro SoC a modo de máster. Además, como ya hemos comentado anteriormente no se va a utilizar el protocolo IP, si no que al igual que el proyecto [20] se realizará una red CAN Bus.

2.4. Conclusión

El apartado de **¡Error! No se encuentra el origen de la referencia.** nos ha mostrado como pese a haber avanzado la tecnología de tal modo que existen tarjetas con microprocesadores que pueden tanto muestrear y codificar las señales de entrada de los sensores como procesarlas y realizar acciones de control con ellas, como se realiza en el proyecto [21] se siguen utilizando los dispositivos DAQ con computadores para el procesador, encareciendo los productos tanto para la empresa que los quiere adquirir como para los clientes. Esto crea una situación en que los barcos más pequeños no puedan tener estos sistemas de monitorización debido al precio disparatado de los dispositivos como al tamaño necesario solamente para almacenarlos (se necesitan tanto cabinas para las tarjetas como consolas para los ordenadores).

Es por ello, que vemos la necesidad de realizar este estudio sobre la viabilidad de cambiar este sistema clásico por tecnología más actual como son los Soc, y más concretamente los dispositivos hardware libre, de bajo coste y bajo consumo tal y como ha comenzado a implantarse en los sectores aeronáuticos [22] e industriales [22].

3. Hardware libe

3.1. Introducción

En este capítulo se va a presentar y explicar detalladamente las características de los dispositivos SoC comerciales más utilizados actualmente.

Comenzaremos dando una visión global de los SoC, más profunda que la que se ha dado en el capítulo 2.1, para continuar con la definición de Open Hardware junto con sus ventajas y desventajas.

Concluiremos el capítulo, tras la comparativa de dispositivos SoC, explicando que módulo es el más indicado para este proyecto y, por tanto, cuál es aquel con el que vamos a trabajar en la implementación del propósito.

3.2. ¿Qué es un SoC?

SoC es el acrónimo de System-on-Chip, y describe la tendencia cada vez más frecuente de usar tecnologías de fabricación que integran todos o gran parte de los componentes de un dispositivo electrónico en un único chip. El diseño de estos sistemas permite una gran integración reduciendo tanto el tamaño del dispositivo como el coste obteniendo además un aumento de rendimiento reduciendo el consumo de energía.

Un SoC está formado por:

- Un microcontrolador o microprocesador (los MPSoC). Es el circuito integral central y más complejo. Se encarga de ejecutar los programas, ejecutando desde sistemas operativos a algoritmos de lenguaje de bajo nivel.
- Controlador de Memoria. El procesador no accede a la memoria RAM de forma directa si no que existe un conjunto de circuitos para controlar los accesos. Normalmente se encuentran en un chip externo al micro.
- Memoria. Incluyendo ROM (memoria de sólo lectura), RAM (memoria de acceso aleatorio), EEPROM (memoria de sólo lectura programable y borrado electrónicamente) y Flash (memorias NAND de acceso muy rápido).

- Chip Gráfico. Chip que se encarga del tratamiento de vídeo liberando de esta tarea al microprocesador.
- Buses. Los buses se encargan de transmitir la información entre los distintos elementos que componen el dispositivo.
- Generadores de frecuencia como osciladores, lazos de seguimiento de fase, PLL.
- Interfaces analógicas incluyendo ADC y DAC.
- Reguladores de voltaje
- Interfaces externos incluyendo estándares como USB, IEEE 1394/Firewire, Ethernet o UART.
- Comunicaciones. Soporte para Wifi y Bluetooth.
- Otros componentes como contadores, temporizadores, relojes a tiempo real y generadores PoR.

3.3. Open Hardware

3.3.1. ¿Qué es?

Open Hardware u Open Source Hardware hace referencia a las especificaciones de diseño de un dispositivo físico que está licenciado de tal manera que ese dispositivo puede ser estudiado, modificado, creado y distribuido por cualquiera.

Open Hardware es, por tanto, una serie de principios de diseño y de prácticas legales con lo cual puede aplicarse a un número de objetos ya sea automóviles, robots u ordenadores.

Al igual que el software open source, el “código fuente” para el *open hardware* – esquemáticos, footprints, diseños lógicos- están disponibles para su modificación por cualquier usuario.

3.3.2. Ventajas y Desventajas

Ventajas

- Proteger y defender la soberanía.

- Fomentar que el hardware sea de calidad, los estándares sean abiertos y mucho más económicos.
- La reutilización y adaptación de diseños permitiendo así innovar y mejorar los diseños de forma colaborativa a nivel mundial.
- Reducir los gastos y tiempos de diseño de las empresas en sus trabajos.
- Crear comunidades de diseños, programación, pruebas y soporte.
- Evitar la alianza *trusted company* y la gestión de derechos digitales[3] que imponen restricciones a los dispositivos electrónicos.

Desventajas

- Los diseños son únicos.
- La persona que quiera utilizar el hardware de otra, primero lo tiene que fabricar, para lo cual tendrá que comprobar los componentes necesarios, construir el diseño y verificar que se ha hecho correctamente. Todo esto tiene un coste en tiempo y dinero.
- Al fabricar un diseño podemos encontrarnos con el problema de la escasez o falta de un material.
- El mundo del hardware está plagado de patentes.
- Debido a las implicaciones que conlleva toda la infraestructura de diseño, simulación, producción e implementación del hardware cualquiera no puede realizar un nuevo dispositivo hardware.

3.4. Comparativa de placas

3.4.1. Arduino

Arduino es una plataforma de electrónica *open source* o de código abierto cuyos principios son contar con software y hardware fáciles de usar. Plataforma muy enfocada a la educación y que limita los pines del microcontrolador, así como las interfaces de programación.

[3] DRM

Software

Si nos centramos en el software, contamos con un entorno IDE multiplataforma, es decir, se puede instalar bajo Windows, Linux y Mac.

¿Qué es un IDE? Es un entorno de desarrollo integrado, es, por tanto, el lugar donde podemos escribir nuestras aplicaciones, descargarlas al Arduino y ejecutarlas o depurarlas desde allí. El entorno es totalmente gratuito y se puede descargar desde la web. También existe desde pocos meses un nuevo IDE online.

El microcontrolador de la placa se programa mediante el lenguaje de programación Arduino, basado en Wiring [23] y el entorno de desarrollo está basado en Processing [24]. Por supuesto, siempre se puede programar directamente usando Eclipse o cualquier otro IDE en C/C++.

Hardware

Hay una infinidad de placas basadas en Arduino. Al ser hardware *open source* cualquiera que desee realizar una nueva placa puede hacerlo. Por ello tenemos Arduino de todos los colores, tamaños y funciones de lo más diverso.

El hardware más sencillo consiste en una placa con un microcontrolador y una serie de puertos de entrada y salida. Los microcontroladores más usados con el Atmega168, Atmega328 y Atmega1280 por su sencillez y bajo coste, aunque también nos encontramos microcontroladores CortexM3 y CortexM0 de 32 bits a 84/48MHz, y placas mixtas que juntas microcontroladores Atmega o ARM con microprocesadores Atheros mucho más potentes.

De todas las placas Arduino que existen en el mercado destacamos dos, Arduino Tian, por ser una solución mixta, poder ejecutar un sistema operativo, tener wifi y bluetooth integrado, 4GB de memoria flash y 64MB de RAM. Por el contrario, no hay GPIOS disponibles del chip Atheros y no hemos encontrado información suficiente sobre la comunicación entre ambos procesadores ni los interfaces de programación compatibles con cada uno de ellos.

También destaca Arduino Due por ser la única que tiene un controlador CAN integrado (aunque en otras soluciones se podrían añadir controladores

externos), y funciona a una frecuencia de 84MHz, 512Kbytes de memoria Flash y 96KBytes de RAM.

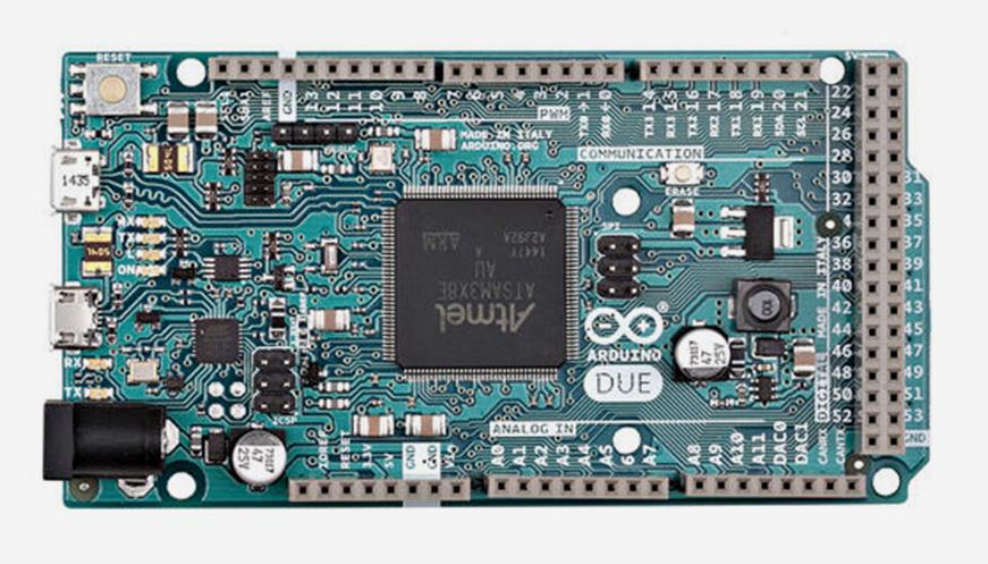

Ilustración 12: Arduino Due. Obtenida de [25]

Todo esto en cuanto a diseños originales de Arduino, como bien se ha mencionado anteriormente, hay una infinidad de diseños no oficiales y modificaciones de los actuales añadiendo cosas, por ejemplo, Industruino [22], que presenta un Cortex M0 a 48MHz, 256kBytes de flash, entradas IDC, pantalla LCD, niveles industriales de I/O, RS485 y un módulo opcional con Ethernet.

Ilustración 13: Industruino. Obtenida de [26]

Si volvemos a Arduino Due (Ilustración 12) ésta está basada en una CPU ARM Cortex M3 siendo la primera placa Arduino en trabajar con 32 bits a una velocidad de 84MHz. Tiene 54 entradas/salidas digitales, pudiendo utilizar 12 como salidas PWM, 12 entradas analógicas, 2 salidas analógicas y 4 puertos serie.

CARÁCTERÍSTICAS TÉCNICAS ARDUINO DUE	
MICROCONTROLADOR	AT91SAM3X8E
TENSIÓN DE TRABAJO	3.3V
TENSIÓN DE ENTRADA	7-12V
PINES DIGITALES I/O	54 (12 PWM)
PINES DE ENTRADA ANALÓGICOS	12
PINES DE SALIDA ANALÓGICOS	2
CORRIENTE POR LOS PINES	130mA
CORRIENTE EN EL PIN DE 3.3V	800mA
CORRIENTE EN EL PIN DE 5V	800mA
MEMORIA FLASH	512KB
MEMORIA RAM	96KB
VELOCIDAD DEL RELOJ	84MHZ
LONGITUD	101.52mm
ANCHURA	53.3mm
PESO	36g

Tabla 1: Características Técnicas Arduino Due. Fuente Propia

Precio

Arduino DUE cuesta unos 36€ más IVA, mientras que Arduino TIAN cuesta 87€ más IVA. El resto de placas descartadas se encuentran entre estos límites. Industruino tiene un precio entre 70 y 130€ según periféricos deseados.

3.4.2. Intel Galileo

Intel Galileo es una variante de placa Arduino, generadas por un acuerdo entre la compañía Intel y Arduino. Junto a esta también tenemos la Arduino Intel Curie, Arduino Intel Edison y Arduino Intel Joule.

Software

Se puede programar desde el mismo IDE de Arduino instalando un plugin gratuito de Intel. Sin embargo, para aplicaciones IoT (la principal aplicación de esta placa) y para explotarla al 100% se debe programar desde el Intel XDK Development. Este programa nos permite programar la placa utilizando Node.js y Phyton aunque las versiones más modernas de este IDE son incompatibles con este modelo de placa [27].

Hardware

Ilustración 14: Intel Galileo. Fuente Propia

Intel Galileo está basada en Intel Quark, una variante de Pentium de 32 bits a una velocidad de 400MHz. El procesador lleva de manera nativa Yocto [28] pero soporta un sistema operativo completo como Linux o Windows.

La placa lleva de serie puerto Ethernet, puertos USB 2.0, lector de tarjetas micro-SD y 20 entradas/salidas digitales (6 de las cuales pueden configurarse como salidas PWM de 8/12 bits).

CARÁCTERÍSTICAS TÉCNICAS INTEL GALILEO	
MICROCONTROLADOR	SoC Quark X1000
TENSIÓN DE TRABAJO	3.3/5V
TENSIÓN DE ENTRADA	7-15V
PINES DIGITALES I/O	14 (6 PWM)
PINES DE ENTRADA ANALÓGICOS	6
MEMORIA FLASH	512KB
MEMORIA RAM	256MB DDR3
ALMACENAMIENTO FLASH	8MB
EEPROM	8kB
VELOCIDAD DEL RELOJ	400MHZ
LONGITUD	124mm
ANCHURA	72mm

Tabla 2: Características Técnicas Intel Galileo

Precio

Se puede encontrar en el mercado español por un precio que ronda los 80€ más IVA, aunque en el mercado norteamericano se vende por 50$.

3.4.3. Raspberry Pi

Con Raspberry Pi tenemos lo que se llama un mini-computador u ordenador de tamaño reducido diseñado por la fundación Raspberry Pi, diseñado y fabricado en UK.

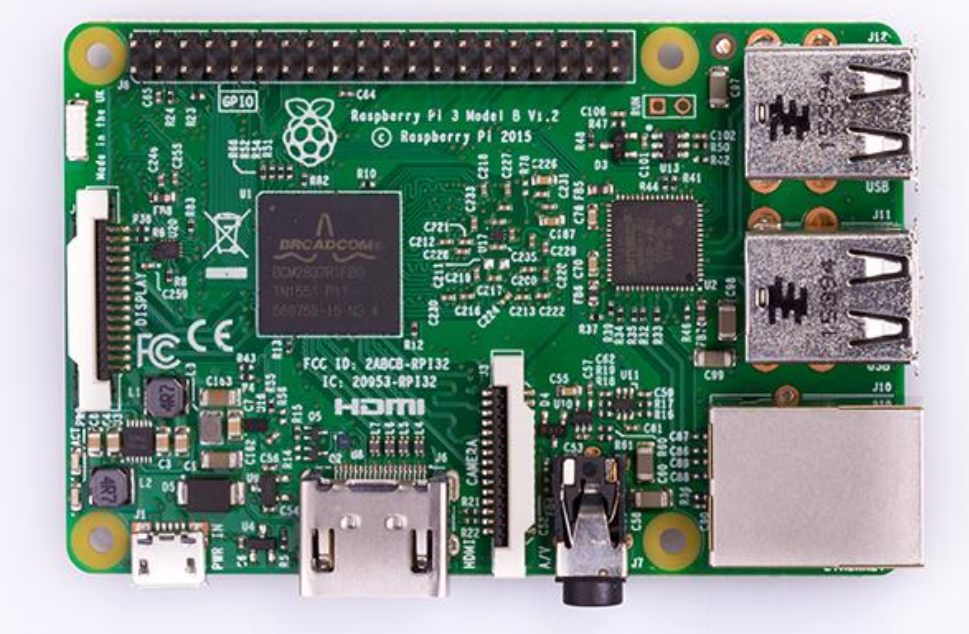

Ilustración 15: Raspberry Pi 3 Modelo B. Obtenido de [29]

Software

Esta es la lista de los sistemas operativos soportados por Raspberry Pi:

- AROS
- Linux
- Plan 9
- Risc OS
- Unix
- Windows 10
- Minibian
- Moebius
- Squeezed Puppy
- WebKiosk
- IPFire
- Micro Elastix
- OpenElec
- OSMC
- Raspbmb
- Xbian

Se puede compilar en cruzado utilizando Eclipse y C/C++. Para la Rpi2, la cual es de 32 bits, existen versiones de SO de tiempo real compatibles, pero a día de hoy no son compatibles con la RPi3.

Hardware

A nivel físico, y centrándonos en la Raspberry Pi 3, modelo B (la versión más nuevo y estándar) presenta un procesador BCM2837 de cuatro núcleos ARM 8 que trabaja con 64 bits y una frecuencia de reloj de 1.2GHz.

Respecto a los protocolos de comunicación, ésta soporta de manera nativa Wi-Fi, Ethernet y BLE.

CARÁCTERÍSTICAS TÉCNICAS RASPBERRY PI 3	
MICROPROCESADOR	ARM Cortex A8
PINES DIGITALES I/O	40
MEMORIA RAM	1GB
VELOCIDAD DEL RELOJ	1.2GHZ
TARJETA GRAFICA	Video Core IV (3D)
VÍDEO	1280X1024 FULL
AUDIO	HDMI
SONIDO	Audio Jack 3.5
INTERFACES	CSI y DSI
CONEXIÓN ETHERNET	10/100

Tabla 3: Características Técnicas Raspberry Pi 3

La Raspberry Pi 3 se puede comprar en 3 modelos: modelo B (el estándar con todas las interfaces de conexión populadas y pines al aire, modelo Zero (mucho

más pequeña, faltando algunas interfaces y pines sin popular, procesador a 1GHz y 512MB RAM), modelo ZeroW (idéntico al modelo Zero pero con antena integrada en PCB y funcionalidad Wi-Fi y BLE), y el modelo Compute Module 3 (de tamaño y conexión SODIMM, con 4GB de memoria eMMC, mismas características que el modelo B pero sin interfaces de conexión, requiere de placa externa y como bonus tiene todos los GPIOs accesibles).

No es perfecta, sus capacidades para reemplazar a microcontroladores son limitadas, el puerto Ethernet no es gigabit y, además el puerto Ethernet comparte bus con el controlador USB, por lo que al usar todo a la vez el rendimiento es menor de lo que podría ser.

Además, a diferencia del resto de modelos, el datasheet del modelo Compute Module no es público y requiere firmar un acuerdo de confidencialidad para obtener más información o trabajar a muy bajo nivel con este diseño.

Precio

El precio varía según el modelo y cantidad: modelo B por 33 € más IVA, modelo Zero 6€ más IVA, modelo Zero W 11€ más IVA, Compute Module 3 por 29€ más IVA y DevKit para Compute Module por 192€ más IVA.

3.4.4. BeagleBone

BeagleBone es una de las rivales directas de Raspberry ya que se trata de otro mini-computador, pero más dedicado a un ámbito profesional con lo cual no tiene una gran comunidad de usuarios desarrollando nuevas funcionalidades para ella. Su principal diferencia es que es una solución mixta, y además del chip similar al de Rpi3[4], tiene 2 microcontroladores, más pines disponibles, 2 controladores CAN [30] y más versiones para elegir.

[4] Raspberry Pi Model 3

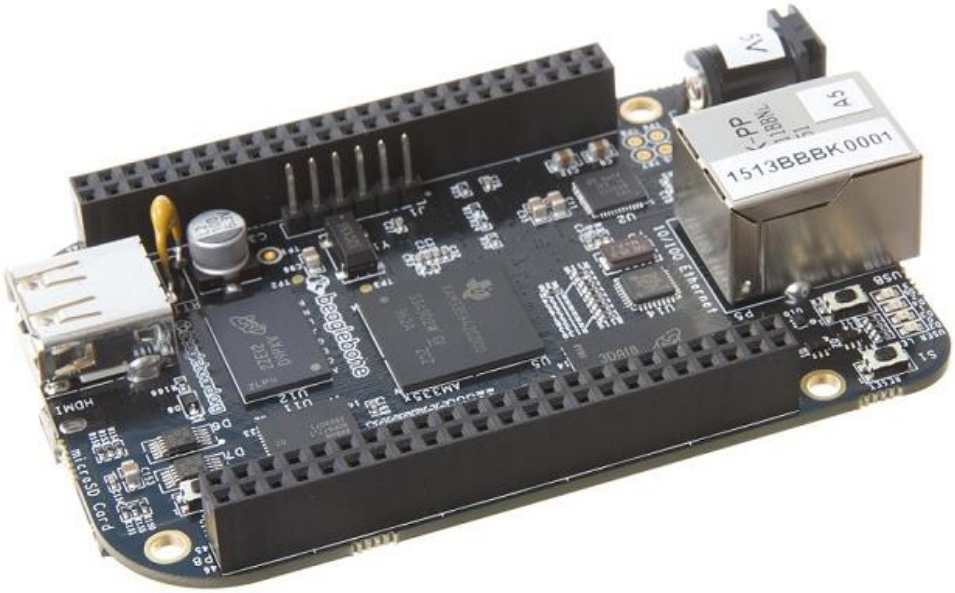

Ilustración 16: BeagleBone Black. Obtenida de [31]

Software

Al tener una comunidad más limitada a día de hoy solo están oficialmente soportados los sistemas Debian, Ubuntu, Android y Cloud9, aunque no manera no oficial hay proyectos que han conseguido hacerlo compatible con otros sistemas operativos.

Hardware

Por un lado, tenemos el modelo Black, Black Wireless y Green, los tres muy parecidos y que describimos a continuación.

CARÁCTERÍSTICAS TÉCNICAS BEAGLEBONE BLACK	
MICROPROCESADOR	ARM Cortex A8
PINES DIGITALES I/O	2 bloques de 46
PINES DE ENTRADA ANALÓGICOS	6
MEMORIA RAM	512MB
MEMORIA INTERNA	4GB
VELOCIDAD DEL RELOJ	1GHZ
TARJETA GRAFICA	SGX530 (3D)
VÍDEO	1280X1024
AUDIO	HDMI
CONEXIONES	Ethernet y 2 USB

Tabla 4: Características Técnicas BeagleBone Black

Por otro lado, está la versión Blue, muy similar, pero con un microcontrolador Cortex M3 extra, funciones wifi, Bluetooth (al igual que el modelo Black Wireless), leds, botones, IMU, barómetro y gestor de baterías.

Por último, está la BeagleBoard-X15, que es el tope de gama, con dos procesadores ARM Cortex-A15 a 1.5GHz, 2GB de RAM, 4GB de Flash, 2DSPs[5], 2 Cortex-M4 y 4 microcontroladores extra que se utilizan para la gestión de la las PRUs[6], Gigabit Ethernet y diversas interfaces de conexión.

[5] Procesador Digital de Señal
[6] Unidad Programable en Tiempo Real [32]

Precio

El precio de BeagleBone Black ronda los 45€ más IVA, si añadimos el bloque Wireless se queda en 66€ más IVA. El modelo Blue se encuentra en el mercado por 75€ más IVA y la BeagleBoard-x15 por 245€ más IVA, aunque a día de hoy no está disponible.

3.4.5. Otras alternativas

Samsung Artik

Toda una serie de placas orientadas al mundo móvil, IoT, almacenamiento y procesamiento en la nube y conexión inalámbrica. Artik 0,5,7 y 10, según periféricos y potencia necesaria. Muy nuevo, sin apenas usuarios de momento ni comunidad, pero hay una gran inversión detrás por parte de Samsung y se observa en la gran cantidad de documentación que se encuentra fácilmente. La más potente, Artik 10, sería la competencia de la Raspberry Pi 3, pero a un precio de casi 150€ y además, según algunas fuentes [33], este producto se va a dejar de lado y no se fabricará más, pidiendo Samsung a los desarrolladores que usen el escalón inferior Artik 7, que también ofrece cosas muy similares a la Rpi3. Este hecho ya genera algo de desconfianza en la plataforma, Cada módulo Artik 7 cuesta unos 50€ y compite directamente con el Rpi3 Compute Module, tanto en funcionalidad como potencia. El DevKit cuesta unos 200€. Todo muy similar a la Raspberry Pi 3 pero aún muy verde.

Nvidia Jetson

Placas muy potentes, con cuatro núcleos ARM A57, GPU Nvidia Maxwell, 4GB RAM, 16GB Flash y mucha conectividad. Son pequeñas y en formato muy similar a Rpi3 Compute Module. Va a salir una versión 2 aún más potente. Problema, su precio es excesivo ya que parte de 400€.

Intel Joule

Solución de Intel para competir con Raspberry Pi 3. Arquitectura Atom Quadcore 64 bit, en dos versiones a 1.5GHz y 1.7GHz, 3 GB de RAM y 8GB de

almacenamiento. Se puede comprar el DevKit para tener interfaces de salida. Bastante más potente que Rpi3, pero mucho más cara, empezando a partir de 200€. Además, es bastante novedosa en el mercado con el cual resulta difícil de adquirir y tenemos una comunidad limitada.

Intel Edison

Solución mixta que junto a un Atom a 500MHz y un Quark a 100MHz, ambos de Intel. Precio en torno a 45-50€ solo por la versión similar a Compute Module, para tener todas las interfaces como Raspberry Pi modelo B hay que comprar otra placa de interfaz. Poca comunidad. Parece cara para lo que ofrece, pareciéndose mucho al Arduino Tian. Potencia insuficiente.

Odroid C2

Producto casi idéntico a Raspberry Pi, mismo procesador a mayor frecuencia, mejor GPU, 2GB de RAM, Ethernet Gigabit y un ADC integrado. Por el contrario, no tiene conectividad inalámbrica, ni Jack de audio. Muy pocos drivers, así como comunidad y documentación limitada. Los vendedores son principalmente asiáticos y al cambio se estaría vendiendo sobre los 45€.

Banana Pi

Línea de productos basada en microprocesadores Allwinner, muy similares en prestaciones y concepto a la Raspberry Pi. Llevan relativamente poco tiempo en el mercado y detrás está la compañía china SINOVOIP de la cual no hay información. Mucha menos documentación y comunidad. Sin embargo, contamos con un gran número de versiones diferentes.

LattePanda

Solución mixta que junta un Intel Atom con un microcontrolador Arduino de 8 bits, mucha memoria y conectividad. Tiene un precio de unos 115€. Sólo lleva un año en el mercado, pero tiene una documentación y comunidad aceptable para iniciar un proyecto utilizando este dispositivo. Detrás está la compañía china DFRobot,

la cual ha sido criticada en el pasado por falta de documentación, abandonar productos sin previo aviso, etc.

Igepv5

Solución mixta muy capaz con 2 cortex A15 a 1.5GHz y 2 Cortez M4 a 200MHz, DSP y GPU. No se ha encontrado ningún establecimiento que lo tenga a la venta, ni comunidad, así como una documentación mínima. No queda claro si se trata de un producto final o un concepto a desarrollar.

PandaBoard

El proyecto parece abandonado y no hay noticias nuevas ni desarrollo desde hace 5 años. No inspira ninguna confianza.

Onion Omega 2 / Chip Pro / ESP32S

Interesantes por ser pequeños, baratos e integrar conectividad inalámbrica, pero demasiado novedosas. No hay garantías de que vayan a salir adelante o tener soporte en un futuro.

3.5. Conclusiones

Se ha mostrado en este capítulo una breve descripción de los modelos más conocidos de placas open hardware a modo de comparativa entre ellas.

Las placas Arduino se descartan por estar demasiado orientadas a la educación, ser poco flexibles, así como microcontroladores insuficientes con poca memoria y poca RAM.

La placa Raspberry Pi se descarta por estar enfocada a otro tipo de proyectos y el sobrecoste que supone el trabajar con CAN Bus, pues que requieren de tarjetas USB to CAN de precios que rondan los 400€ como la tarjeta IXXAT.

Las placas BeagleBone y BleagleBoard son muy interesantes por ser soluciones mixtas que integran microcontroladores además del procesador principal y 2 controladores CAN, sin embargo de momento se han descartado por falta de documentación, una comunidad de usuarios muy reducida y ahora mismo están

en un estado incierto, con muchas versiones de cada placa, placas nuevas que aún no están a la venta ni validadas, y pocas garantías de que va a pasar con las versiones existentes en cuanto a soporte y disponibilidad.

Se han estudiado brevemente otras opciones de SoCs, pero se han descartado por diversas razones, cómo se indica justo en el apartado anterior. Algunas de ellas se han descartado por ser demasiado nuevas y tal vez en unos meses/años se conviertan en productos estables e interesantes. Aquí merece la pena destacar dos opciones: las opciones que usan procesadores Intel Atom (Intel Joule, LattePanda e Intel Edison) que a medio plazo pueden ser una alternativa fuerte a ARM; y la familia de productos Samsung Artik que tiene la ventaja de ofrecer una gama de productos desde simples microcontroladores hasta más orientados a la nube.

La opción que parece más interesante ahora mismo es la plataforma Intel Galileo. Es potente, buena relación calidad/precio y garantía de que va a seguir habiendo soporte y se va a seguir fabricando. Además, es compatible con todos los dispositivos diseñados para ser utilizados junto a Arduino Uno contando así con la comunidad Arduino como soporte.

4. Protocolo de comunicación CAN BUS

En los siguientes capítulos vamos a tratar los fundamentos teóricos de dos protocolos de comunicación: el protocolo CAN Bus y el protocolo Ethernet.

El motivo de la elección de estos dos protocolos de comunicación es que el protocolo CAN Bus, como ya vimos en el capítulo 2.2 es el más utilizado actualmente en el sector naval, con lo cual es el primero que se debe tener en cuenta a la hora de realizar un diseño aplicado a este sector.

Se ha elegido también estudiar teóricamente Ethernet debido a que en los últimos años en el sector industrial se está abandonando el cableado CAN por cableado Ethernet debido a que éste último es más seguro frente a interferencias, alcanza mayores velocidades de transmisión y es más económica su instalación.

4.1. Introducción

CAN (Controller Area Network) es un estándar tipo bus multi-master pensado para conectar unidades de control electrónico ECU (del inglés Electronic Control Unit), conocidos comúnmente como nodos, y permitir la comunicación entre ellos sin necesidad de un ordenador host reduciendo así cableado.

El bus CAN es un protocolo serie asíncrono del tipo CSMA/CD (Carrier Sense Multiple Access With Collision Detection). Debido a este tipo de portadora, cada nodo de la red debe monitorizar el bus y si detecta que no hay actividad entonces envía los mensajes. Al presentar detección de colisiones, si dos nodos de la red comienzan a transmitir a la vez un mensaje, ambos detectan la colisión. Además, es un protocolo multicast, es decir, todos puedes hablar y escuchar.

Se utilizan cables trenzados para crear un cable diferencial con alta inmunidad a las interferencias electromagnéticas, y en algunos casos, además van apantallados. La impedancia característica de la línea es del orden de 120Ω, por lo que se emplean impedancias de este valor en ambos extremos para evitar que las ondas se reflejen y acabar teniendo una antena en lugar de un bus de comunicación. Este bus tiene una longitud máxima de 1km a 40kbps y una velocidad máxima de 1Mbps a 40 metros.

4.2. Capa Física

El protocolo CAN define la capa de enlace de datos y parte de la capa física basándose en el modelo OSI. Sin embargo, la ISO define un estándar que incluye el protocolo CAN más una parte extra perteneciente a la capa física. Por este motivo, los siguientes puntos están basados en el estándar ISO.

4.2.1. Codificación de Bit

Las tramas de los mensajes que se transmiten mediante este estándar se codifican mediante el código NRZ (Non-Return-to-Zero).

Este método es un código binario en el que los unos se representan normalmente por un voltaje positivo, mientras que los ceros se representan por un voltaje negativo o un voltaje nulo. Además, no tenemos flancos de subida o bajada en los bits debido a que no se da un voltaje nulo tras cada voltaje positivo o negativo, como sucede en el código RZ (Return-to-Zero).

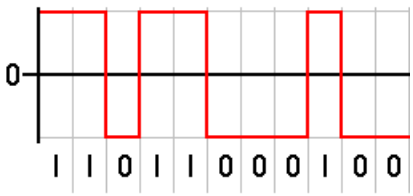

Ilustración 17: Trama NRZ. Obtenida de [34]

En la codificación NRZ, el nivel de señal permanece constante (tal cual se observa en la Ilustración 17) durante el tiempo de bit y por tanto solo un tiempo es requerido para la representación de un bit. El nivel de la señal puede permanecer constante durante un gran periodo de tiempo si tenemos varios bits seguidos de la misma polaridad, entonces, las medidas se deben tomar asegurándose que el intervalo máximo entre dos límites de señal no se excede.

Debido a esta inexistencia de flancos en el bit, para poder resincronizar la señal cuando se transmiten un gran número de bits consecutivos con la misma polaridad se debe utilizar el procedimiento de violación digital (bit stuffing) para asegurar la sincronización de todos los nodos.

Este método inserta dos veces un bit sin información, que mantiene la transmisión siempre que tengamos 5 bits consecutivos de la misma polaridad. Si la trama es de unos, se insertan ceros, si la trama es de ceros se inserta un bit de polaridad positiva y un bit de polaridad negativa.

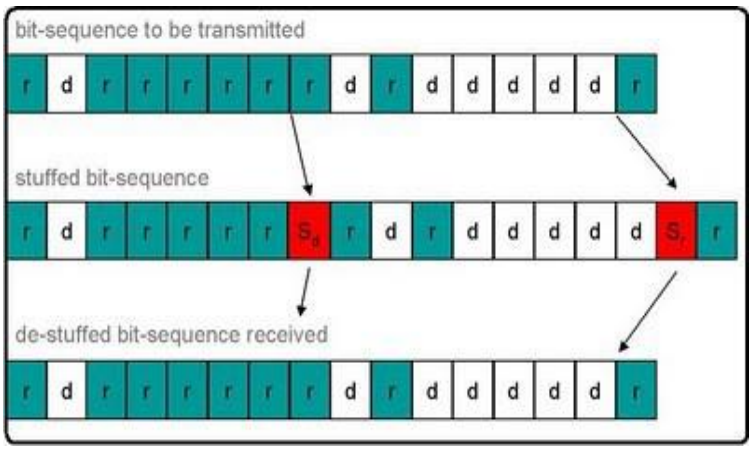

Ilustración 18: Bit Stuffing. Obtenida de [35]

A primera vista, la relación entre los bits de datos y los bits rellenados se puede estimar como 5 a 1, pero en el peor de los casos, el máximo número de bits rellenados que se pueden insertar en una trama se puede calcular como:

$$S_{max} = \frac{n-1}{4}$$

Donde n es el número de bits de datos.

Debido a que el campo de control (Control Field) de una trama CAN contiene dos bits reservados que toman el valor dominante, el valor máximo de bits rellenados que toman el valor dominante, el valor máximo de bits rellenador no siempre es el mismo que el obtenido teóricamente con la fórmula anterior, para ello podemos observar la siguiente tabla:

Data Length Code	Máximo Estimado	Número de mensajes CAN existentes con:			
DLC	s_max	s_max	s_max-1	s_max-2	s_max-3
0	8	0	0	1	13
1	10	0	0	0	30
2	12	0	0	0	32
3	14	0	0	5	444
4	16	0	0	2	>171
5	18	0	0	0	>82
6	20	0	0	1	>175
7	22	0	1	124	>15205
8	24	0	0	21	>2660

Tabla 5: Cantidad Máxima de bits de relleno. Fuente propia.

4.3. Tiempo de Bit y Sincronización

4.3.2. Tiempo de Bit

En el estándar CAN, un tiempo de bit equivale a 4 segmentos de tiempo que no se superponen:

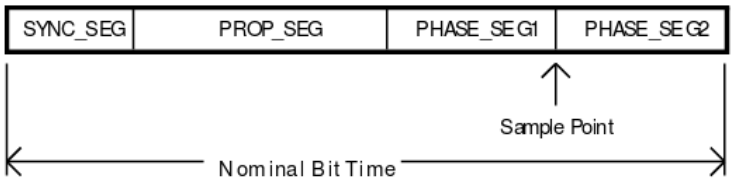

Ilustración 19: Segmento de Bit. Fuente Propia

- SYNC_SEG: Presenta una duración de 1 Quantum[7]. Se utiliza para sincronizar los nodos bus.
- PROP_SEG: Se puede programar para que tenga una duración entre 1 y 8 Quantum. Se usa para compensar los retrasos de la señal.
- PHASE_SEG1: Se puede programar para que tenga una duración entre 1 y 8 Quantum. Se usa para compensar los errores de cambios de fase y puede ser medida durante la resincronización.
- PHASE_SEG2: Es el máximo entre PHASE_SEG1 y el tiempo de procesamiento de la información (≤ 2 Quantum9. Tiene la misma funcionalidad que PHASE_SEG1.

La asociación CiA [36] (CAN in Automation) ha realizado una tabla que asocia la longitud del enlace con los quantum y el baudaje. Estos valores son recomendados para usar en redes de propósito general. Además, recomienda un tiempo de bit y la posición del punto de muestra para que los nodos en cada de que sean de diferentes proveedores puedan conectarse sin necesidad de realizar cálculos extras.

[7] Los Quantum son la medida temporal más pequeña que usa un nodo CAN. Se generan mediante la división de la frecuencia del oscilador de cada nodo CAN. Cada bit puede tener una duración entre 8 y 28 Quantum.

Velocidad	Tiempo de bit	Quantum por bit	Longitud del Quantum	Punto de Muestra	Baudaje 1 (16MHz)	Baudaje 2 (16MHz)
1 Mbit/s	1 µs	8	125 ns	6 tq	00 h	14 h
800 kbps	1.25 µs	10	125 ns	8 tq	00 h	16 h
500 kbps	2 µs	16	125 ns	14 tq	00 h	1C h
250 kbps	4 µs	16	250 ns	14 tq	01 h	1C h
125 kbps	8 µs	16	500 ns	14 tq	03 h	1C h
50 kbps	20 µs	16	1.25 µs	14 tq	09 h	1C h
20 kbps	50 µs	16	3.125 µs	14 tq	18 h	1C h
10 kbps	100 µs	16	6.25 µs	14 tq	31 h	1C h

Tabla 6: Relación entre velocidad y baudaje. Fuente propia.

4.3.3. Sincronización

En el nivel de bit, CAN utiliza la sincronización para la transmisión de los bits. Esto mejora la capacidad de transmisión, pero también hace que se requiera un método sofisticado.

Mientras que la sincronización de bit en una transmisión orientada al carácter (asíncrona) se realiza sobre la recepción del bit de inicio disponible en cada carácter, un protocolo de transmisión síncrono tiene solo un bit de inicio disponible al principio de cada paquete. Para permitir que el receptor lea correctamente los mensajes, se debe hacer una continua resincronización. Segmentos de fase de amortiguamiento se insertan, por lo tanto, antes y después del punto de muestreo nominal dentro del intervalo de bit.

El protocolo CAN regula el arbitraje bit-wise. La señal se propaga desde el emisor al receptor y vuelve al emisor, completándose este ciclo en un tiempo de bit. Para propósitos de sincronización, se complementa de un segmento de tiempo más, el segmento de retraso de propagación, que es necesario añadir al tiempo reservado para la sincronización, el segmento de fase. El retraso de propagación se tiene en cuenta la propagación de la señal en el bus, así como los retrasos causados por los nodos de transmisión y recepción.

Se pueden distinguir dos tipos de sincronización: la sincronización fuerte al principio del paquete (el tiempo de bit se reinicia al final del segmento de sincronización) y la resincronización con el paquete.

4.4. Propagación de la señal

Debido tanto a la línea bus como a los circuitos electrónicos de los nodos se producen retrasos en la propagación en la señal, los cuales nos vemos obligados a tratar de compensar para que la comunicación sea efectiva.

La suma del retraso del controlador, aislamiento galvánico (en caso de que se esté usando), transceptor y la línea bus tiene que ser menor que la longitud del PROP_SEG. Hay que tener en cuenta que el retraso del controlador suele ser de 50 a 62 ns, la del optoacoplador de 40 a 140 ns, la del transceptor de 120 a 250 ns y el cable sobre los 5 ns/m. Estos retrasos se tienen que contabilizar de manera doble debido a que después de la sincronización, el nodo más lejano sigue esperando cambios en la señal de bit con retraso. Por tanto,

$$t_{propagacion} = 2 \cdot \left(t_{cable} + t_{controlador} + t_{optoacoplador} + t_{transceptor}\right)$$

Teniendo en cuenta la fórmula anterior, podemos asumir que la velocidad de propagación está inversamente relacionada con la longitud de la línea bus y que si queremos altas velocidades nos tendremos que limitar a líneas de comunicación muy cortas.

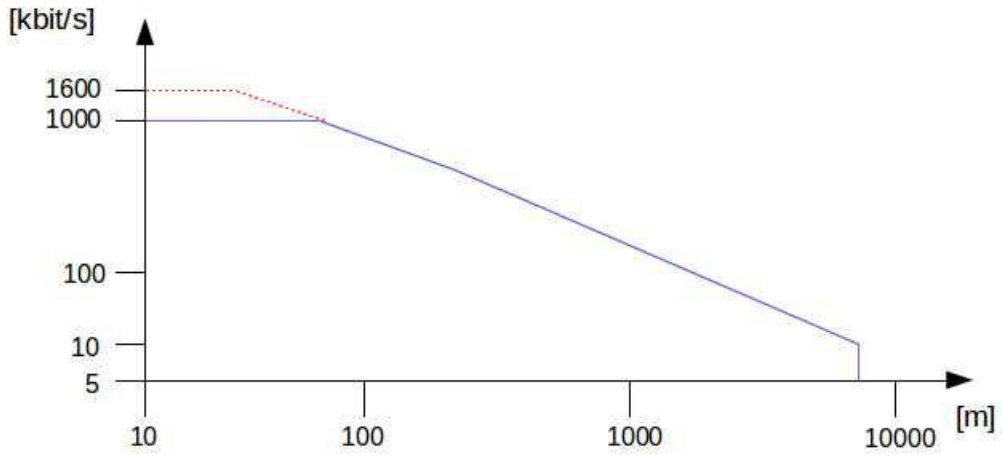

Ilustración 20: Relación entre velocidad y Longitud. Fuente Propia

Hay que tener en cuenta que la máxima longitud de línea bus en una red CAN está determinada por los siguientes factores:

- Los retrasos en los nodos y en las líneas.
- Las diferencias del tamaño del quantum debido a la tolerancia relativa del oscilador.
- Caídas en la amplitud de la señal debido a la impedancia del cable y los nodos.

Para facilitar los cálculos, la norma ISO 11898 propuso la siguiente tabla que relaciona velocidades de propagación con el tamaño máximo que puede alcanzar el cable. Estos valores son teóricos y se realizaron tomando en cuenta transceptores que cumplen esta norma y sin la aparición de optoacopladores, por lo que en un diseño final se ve relativamente reducida.

Velocidad	Longitud	Tiempo de Bit
1 Mbit/s	30 m	1 µs
800 kbps	50 m	1.25 µs
500 kbps	100 m	2 µs
250 kbps	250 m	4 µs
125 kbps	500 m	8 µs
62.5 kbps	1000 m	20 µs
20 kbps	2500 m	50 µs
10 kbps	5000 m	100 µs

Tabla 7: Longitud máxima de línea bus. Fuente Propia

4.5. Resincronización

Una red CAN consiste en una serie de nodos, cada uno de los cuales sincronizado mediante su propio reloj. Debido a esto se producen cambios de fase en cada uno de los nodos. El controlador CAN proporciona un mecanismo de sincronización que compensa los cambios de fase mientras recibe cada trama CAN.

El funcionamiento de la sincronización funciona tal que el controlador CAN espera un cambio de polaridad durante la recepción de la subtrama SYNC_SEG. En caso de haber un retraso y detectar el cambio de polaridad durante la recepción de la subtrama PROP_SEG, el receptor alarga la duración de la subtrama PHASE_SEG1 con el máximo valor programado en el campo RJK (Resynchronization Jump Width).

Si tenemos el caso contrario, una transición más rápida, y detectamos el cambio de polaridad durante la subtrama PHASE_SEG2 entonces el receptor acorta la duración de este segmento restándole el máximo valor programado en el campo RJW.

4.6. Medio físico

Existe una serie de estándares oficiales que versan sobre los medios portadores CAN. El más importante para aplicaciones de propósito general es el estándar de alta velocidad ISO 11898 y, por tanto, es el que vamos a seguir para explicar los siguientes puntos de este apartado.

El estándar ISO 11898-2 dice que el cableado de la red CAN debe ser una estructura de una sola línea para minimizar los efectos de la reflexión a través de la línea Bus. Además, esta línea debe terminar con resistencias a ambos lados.

Debido a la naturaleza diferencial de la transmisión de una señal CAN es insensible a interferencia electromagnéticas si la caída de tensión a ambos lados de la línea es constante.

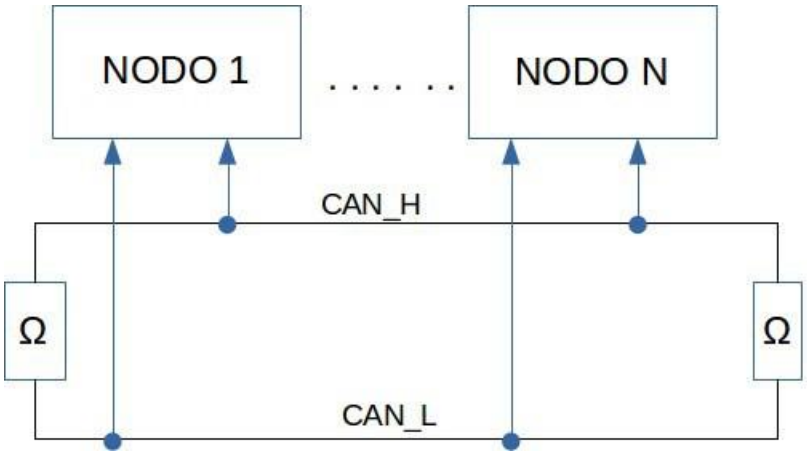

Ilustración 21: Cableado básico de una red. Fuente Propia

Cada nodo (siempre según el estándar ISO 11898) requiere un microcontrolador y un controlador CAN que se conecta al transceptor vía línea serie al puerto de salida (Tx) y al de entrada (Rx). El voltaje de referencia Vref proporciona un

voltaje de salida de la mitad de la tensión nominal Vcc. A su vez el transceptor se debe alimentar a 5V.

En condiciones estáticas la tensión de entrada diferencial de un nodo está determinada por la corriente que fluye a través de la resistencia de la entrada diferencial de dicho nodo. En caso de querer enviar un bit dominante, las salidas de los transistores del nodo transmisor están conmutados causando un flujo de corriente mientras que los transistores están en abierto para emitir un bit recesivo.

Los nodos detectan una condición recesiva en el bus si el voltaje del CAN_H no es mayor que el voltaje del CAN_L más un 0.5V. Si el voltaje del CAN_H es como mínimo 0.9V superior al voltaje del CAN_L, entonces el nodo detectará que en el bus hay una condición dominante.

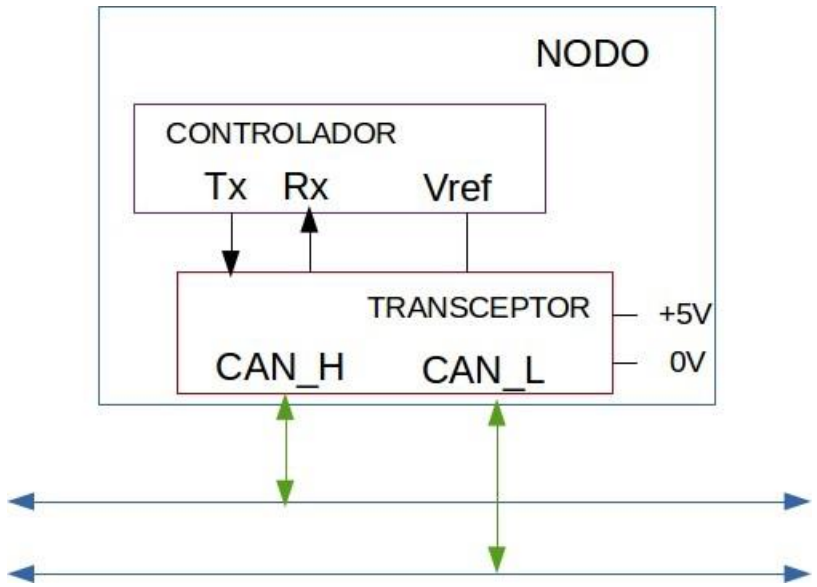

Ilustración 22: Detalle de un nodo CAN. Fuente Propia

El voltaje nominal del estado dominante suele ser 3.5V para la línea CAN_H y 1.5V para la línea CAN_L.

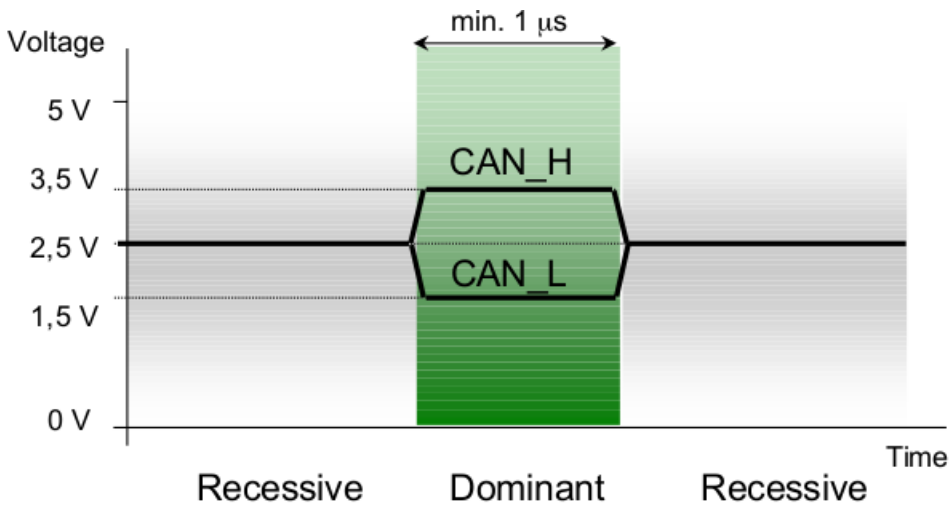

Ilustración 23: Niveles de Bit. Obtenido de [37]

Como se ve en la Ilustración 22: Detalle de un nodo CAN. Fuente Propia, el controlador está conectado al transceptor vía los puertos Tx y Rx que como hemos comentado anteriormente son los puertos de salida de datos y de entrada de datos respectivamente, de una línea serie. El transceptor está conectado a la línea bus vía sus dos terminales CAN_H y CAN_L los cuales aportan la capacidad diferencial de transmisión y recepción al sistema.

4.7. Criterios de Diseño

Existen tres criterios básicos de los cuales vamos a partir para definir el diseño de una red basada en CAN.

Cada módulo que forme parte de la red debe soportar como mínimo una velocidad de 20kbps.

- Si la longitud del cable va a ser superior a 200 m se deben utilizar optoacopladores.
- Si la longitud del cable va a ser superior a 1000 m se deben colocar repetidores o puentes.

4.7.1. Características del cable CAN

De acuerdo a la norma ISO 11898-2, los cables que se pueden utilizar como líneas CAN Bus deben tener una impedancia nominal de 120Ω y el retraso

nominal de 5ns/m. La terminación de la línea se tiene que realizar mediante resistencias de 120Ω colocadas en ambos finales de línea.

La resistencia del cable debe ser 70mΩ/m.

Para intentar minimizar esta impedancia extra que aportan los cables en función de su longitud, nos interesa que, para tramos largos, estos cables tengan resistencias por metro mucho menores que las que aplica la norma ISO. Además, al ser mayores longitudes, si queremos mantener constante el voltaje diferencial para tener una red inmune a interferencias electromagnéticas tendremos que poner resistencias finales mucho mayores de 120Ω.

Longitud Bus	Cable Bus		Resistencia de Terminación
	Impedancia	Sección del cable	
0 ~ 40 m	70 mΩ/m	AWG23-AWG22	124Ω (1%)
40 ~ 300 m	<60 mΩ/m	AWG22-AWG20	127Ω (1%)
300 ~ 600 m	<40 mΩ/m	AWG20	150 ~ 300Ω
600 m ~ 1 km	<26 mΩ/m	AWG18	150 ~ 300Ω

Tabla 8: Características Cable CAN. Fuente Propia

Hay que tener cuidado con la Tabla 8 ya que no se han tenido en cuenta los conectores DB9 para conectar los nodos a las líneas bus. Para poder hacer un cálculo más apurado hay que sumar por cada nodo entre 5mΩ y 20mΩ a la resistencia total de la transmisión.

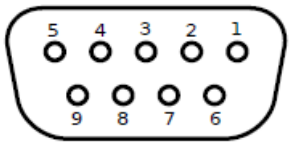

Ilustración 24: Conector DB9. Obtenido de [38]

Las conexiones que van a cada uno de los pines del conector de la Ilustración 24 son los siguientes:

PIN	SEÑAL	DESCRIPCIÓN
1	-	Pin Reservado
2	CAN_L	Línea CAN_L
3	CAN_GND	Tierra del CAN
4	-	Pin Reservado
5	CAN_SHLD	Conexión del CAN Shield (Opcional)
6	GND	Tierra externa opcional
7	CAN_H	Línea CAN_H
8	-	Pin Reversado
9	CAN_V+	Alimentación Externa (Opcional)

Tabla 9: Asignación de Pines. Fuente Propia

4.7.2. Topologías

La topología de una red CAN debe ser lo más similar posible a una estructura de una única línea para evitar los reflejos de las ondas. Esencialmente depende de los parámetros del tiempo de bit, la longitud troncal del cable Lt y la longitud del tramo de caída de los nodos Ld para saber si toleramos las reflexiones o no.

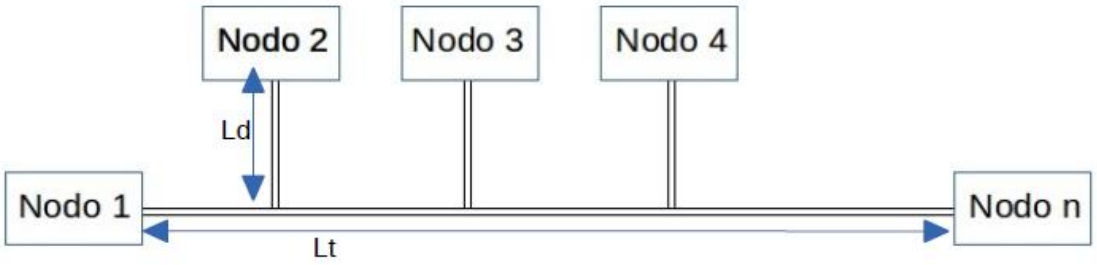

Ilustración 25: Topología Ideal. Fuente Propia

En la práctica para conectar los nodos se utilizarán pequeñas líneas de transmisión stub Ld necesariamente. Tienen que ser lo más cortas posibles, especialmente a altas velocidades de transmisión, por ejemplo, a 1 Mbps la longitud de Ld no será mayor de 30 cm.

Existen unas reglas para calcular la máxima longitud del cable Ld y la longitud acumulativa de Ld (Ldi).

$$Ld < \frac{t_{propseg}}{(50 \cdot tp)}$$

$$\sum_{i=1}^{n} Ldi < \frac{t_{propseg}}{(10 \cdot tp)}$$

Siendo $t_{propseg}$ la longitud del segmento de propagación y tp el retraso de la línea.

Ejemplo:

Tenemos una velocidad de transmisión de 500kbps. Calculamos $t_{propseg}$ asumiendo que tiene una duración de 6 quantum:

$$t_{propseg} = 6 \cdot 125\ ns = 750\ ns$$

El valor de 125 ns sale de la duración de cada quantum (ver Tabla 6). También sabemos que:

$$tp = 5\ ns/m$$

Aplicando las fórmulas anteriores tenemos que:

$$Ld < \frac{750\ ns}{50 \cdot 5\ ns/m} = \frac{750}{250} m = 3m$$

$$\sum_{i=1}^{n} Ldi < \frac{750\ ns}{10 \cdot 5\ ns/m} = \frac{750}{50} m = 15\ m$$

En resumen, lo máximo que puede medir cada sublínea Ld es de 3 metros y la suma de la medida de todas las sublíneas de la red no puede superar los 15 metros.

Topología de resistencias divididas

Esta variante de la topología ideal presenta unas características mejoradas frente a las interferencias electromagnéticas sin modificar las características del cable bus ya que las frecuencias no deseadas se filtran.

Esto se consigue con un condensador acoplado entre dos resistencias de valor la mitad del valor de la resistencia final de la topología ideal (por ejemplo, si es la topología ideal son 124Ω, se pondrían 2 resistencias de 62Ω) que acoplan los ruidos que se encuentran a altas frecuencias a la tierra, distinta al pin del conector, funcionando como un típico filtro paso bajo.

$$fc = \frac{1}{(2 \cdot \pi \cdot Rt \cdot Cg)}$$

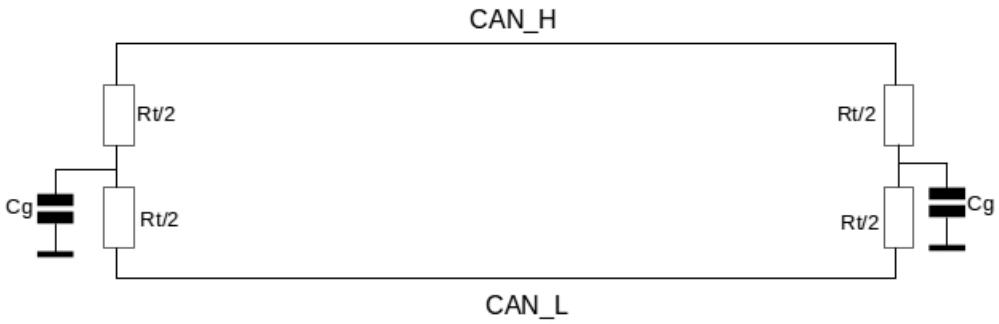

Ilustración 26: Topología Split. Fuente Propia

Hay que tener cuidado a la hora de elegir las resistencias porque si no coinciden sus impedancias se reduce el efecto de inmunidad electromagnética que tiene presente esta topología.

Un valor típico de condensador para trabajar con CAN a altas velocidades es de 4.7nF, con el que se genera una caída de 3dB a 1.1Mbps, pero, como es lógico este valor depende de la línea con la que se esté trabajando.

Topología de múltiples terminaciones

Esta topología está pensada para estructuras que difieren de la topología ideal por ejemplo por tener caídas de cable a los pocos metros. Esta topología se puede utilizar conjuntamente con la topología de resistencias divididas.

Esta topología consiste en distribuir la resistencia de finalización total en más de dos resistencias. Si por ejemplo una topología en estrella con tres ramas, entonces hay que terminar cada banda con aproximadamente tres veces el total de la resistencia de finalización. Hay que tener cuidado en no sobrepasar el límite de impedancia de salida del transceptor.

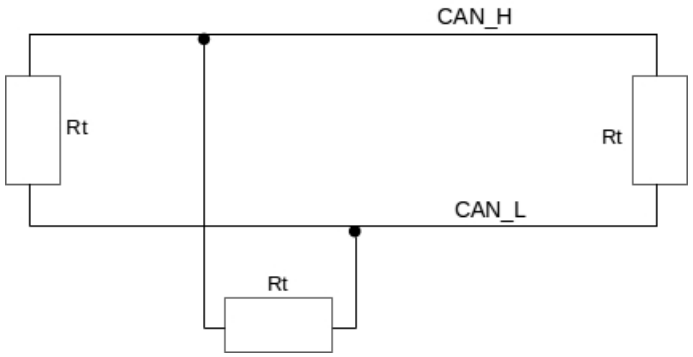

Ilustración 27: Topología con Múltiples Terminaciones. Fuente Propia

Ejemplo:

Partimos de una topología ideal con una resistencia de terminación de 120Ω y queremos realizar la topología de la Ilustración 27 que presenta 3 ramas y queremos calcular la resistencia de termino de cada rama. Entonces:

Calculamos el total de la nueva resistencia de término:

$$R' = 120 \cdot 3 = 360\ \Omega$$

Para saber el valor exacto que colocar en cada rama, dividimos R' entre dos como si se tratara de una topología ideal:

$$Rt = \frac{360}{2} = 180\ \Omega$$

Topología patentada en estrella

Daimler-Benz patentó una topología en estrella con una terminación común en forma de estrella. Esta topología es resistente ante problemas de resonancia y armónicos no deseados, y es insensible ante señales que causan interferencias.

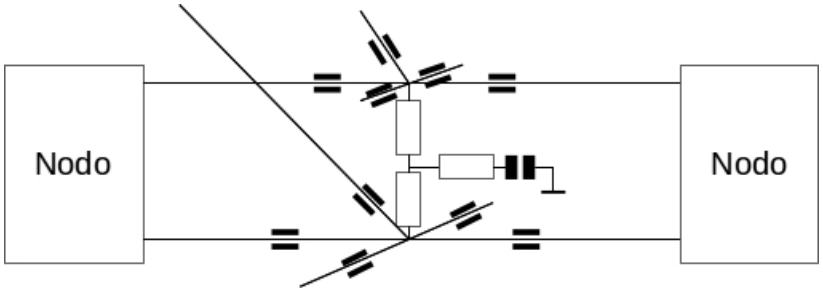

Ilustración 28: Topología Patentada en Estrella. Fuente Propia

Las resistencias en paralelo toman el valor de 30Ω, la resistencia en serie con el condensador toma el valor de 10Ω y el condensador 15 nF.

Los núcleos de ferrita son equivalentes a un paralelo de una resistencia de 400 Ω, una bobina de 2μH y un condensador de 10pF.

Para entender porque hay una sola terminación, hay que notar que una línea de transmisión con una terminación única es suficiente para evitar las resonancias porque la onda que viaja a través será absorbida por el punto de terminación y nunca se reflejará.

4.8. Nivel de Enlace

4.8.1. Principio de Intercambio de Datos

CAN está basado en un protocolo de transmisión orientado al mensaje por tanto funciona mediante un mecanismo de multidifusión, definiendo el mensaje en lugar de las direcciones de los nodos. Cada mensaje tiene un identificador, que es lo único en toda la red, que define el contenido y prioridad del mensaje que sirve como arbitraje del bus cuando varios nodos compiten por el acceso.

Como resultado del direccionamiento orientado al contenido, se obtiene un alto grado de flexibilidad en la configuración del sistema. Es fácil añadir nodos a una red existente y se puede hacer sin necesidad de modificar ni el hardware ni el software de los nodos existentes, siempre y cuando los nodos añadidos sean puramente receptores. Esto permite un diseño modular, la recepción múltiple de datos y sincronizar procesos distribuidos. Además, la transmisión de datos no se basa en la disponibilidad de tipos específicos de nodos, con lo que se complica y mejora la red.

4.8.2. Transmisión de Datos en Tiempo Real

En el proceso en tiempo real, la urgencia con la cual los mensajes deben ser intercambiados a través de la red puede diferir. La prioridad a la cual un mensaje se transmite comparado con otro mensaje menos urgente, se especifica mediante el identificador de cada mensaje. Las prioridades se establecen durante el diseño del sistema en forma de valores de correspondencia binaria y no se pueden cambiar dinámicamente. El identificador con el menor número binario tiene la prioridad más alta.

Los conflictos de acceso al bus se resuelven por el arbitraje de bits en los identificadores que participan en cada nodo observando el nivel de bit del bus. Esto ocurre de acuerdo con el mecanismo y el cable, mediante el cual el estado dominante sobrescribe el recesivo. Todos los nodos con transmisión recesiva y observación dominante pierden el acceso al bus. El resto se convierten automáticamente en los receptores del mensaje con la más alta prioridad y no intenta la transmisión hasta que es el bus el que está disponible de nuevo.

Las solicitudes de transmisión se manejan en orden de importancia para el sistema como un todo. Esto resulta especialmente ventajoso en situaciones de sobrecarga. Dado que en el acceso al bus se prioriza en base a los mensajes, es posible garantizar una baja latencia individual en sistemas de tiempo real.

4.8.3. Formato de trama

El protocolo CAN soporta dos formatos de trama, con la única diferencia de la longitud de sus identificadores. La trama básica soporta una longitud de 11 bits y la trama extendida soporta 29 bits.

Formato de Trama Básica

Un formato de trama básica empieza con un bit de inicio llamado SOF (Start Of Frame), seguido por el campo de arbitraje que consiste en el identificador y el bit RTR (Remote Transmission Request), utilizado para distinguir entre una trama de datos normal y una trama de petición de datos llamada trama remota. El siguiente campo de control contiene el IDE (Identifier Extension), bit que distingue entre una trama básica o extendida, seguido del DLC (Data Length Code*)* utilizando para indicar el número de bytes de datos. Si el mensaje es una trama remota, el campo DLC contendrá el número de bytes de datos solicitados. El campo de datos que sigue puede contener hasta 8 bytes. La integridad de la trama está garantizada a través de la suma de CRC (Cyclic Redundant Check). El campo ACK comprende la ranura ACK y el delimitador ACK. El bit de la ranura ACK se envía como recesivo y se sobrescribe como dominante por dichos receptores, que tienen en este momento los datos recibidos correctamente. Los mensajes incorrectos son reconocidos por los receptores, independientemente del resultado de la prueba de aceptación. El final del mensaje llega con el EOF (End Of Frame). El espacio entre trama IFS (Space Frame Intermission) es el número mínimo de bits que separan mensajes consecutivos. A menos que otra estación se ponga a transmitir, el bus permanece inactivo.

Ilustración 29: Formato de Trama básica

Formato de Trama Extendida

Como hemos comentado anteriormente, la diferencia entre una trama extendida y una trama básica es la longitud de identificador utilizado. El identificador de 29 bits se compone de 11 bits y una extensión de 18 bits. La distinción entre el formato de trama extendida y la básica se realiza mediante el bit IDE, que se transmite como dominante en el caso de una trama de 11 bits, y se transmite como recesivo en una de 29. Como los formatos tienen que coexistir en el bus, se establece cuál es el de mayor prioridad, en el caso de colisión entre tramas de diferente formato y el mismo identificador básico/identificador. Los mensajes de 11 bits tienen prioridad frente al de 29 bits.

El formato extendido tiene algunas ventajas y desventajas. El tiempo de latencia en el bus es más largo (mínimo dura 20 tiempos de bit), los mensajes en formato extendido requieren más ancho de banda (aproximadamente un 20%), y el rendimiento de detección de error es menor debido a que el polinomio elegido para el CRC de 15 bits está optimizado para tramas de hasta 112 bits.

Los controladores CAN que soportan formatos de trama extendida, también son capaces de enviar y recibir tramas básicas. Los controladores CAN que solo admiten tramas básicas, no pueden reconocer tramas extendidas correctamente. Sin embargo, existen controladores CAN que solo soportan tramas básicas pero que reconocen las tramas extendidas solo que las ignoran.

4.8.4. Detección de errores

A diferencia de otros sistemas de bus, el protocolo CAN no utiliza mensajes de reconocimiento, si no que señala los errores de inmediato a medida que ocurren. Para la detección de errores el protocolo CAN implementa tres mecanismos a nivel de mensaje:

- Comprobación de Redundancia Cíclica (CRC): El CRC salvaguarda la información en la trama añadiendo una trama de comprobación de secuencia (FCS) al final de la transmisión. En el receptor este FCS se recalcula y prueba contra el FCS recibido. Si no coinciden, se produce un error.

- Comprobación de trama: Este mecanismo verifica la estructura de la trama transmitida mediante la comprobación de los campos de bits contra el formato fijo y el tamaño de la trama. Los errores detectados por los controles de trama se denominan errores de formato.
- Errores ACK: Los receptores de un mensaje responden las tramas recibidas con un ACK. Si el transmisor no recibe el ACK, ocurre un error de ACK.

El protocolo CAN también implementa dos mecanismos para la detección de errores a nivel de bit:

- Monitorización: La capacidad del transmisor para detector errores se basa en la monitorización de señales del bus. Cada nodo que transmite observa el nivel del bus y así detecta diferencias entre el bit enviado y el bit recibido. Esto permite la detección segura de errores globales y errores locales en el transmisor.
- Rellenado de Bits: La codificación de cada bit individual es probada a nivel de bit. La representación de bits utilizada en CAN es, como hemos comentado anteriormente, la NRZ. La sincronización de extremos se genera mediante el relleno de bits.

Si uno o más errores se descubren por lo menos en un nodo utilizando los métodos anteriores, la transmisión se interrumpe mediante una trama de error. Esto evita que otros nodos acepten el mensaje y así garantizar la coherencia de los datos en toda la red. Después de la transmisión de un mensaje erróneo que se ha anulado, el remitente de forma automática reintenta la transmisión de manera automática pudiendo los nodos competir de nuevo por el acceso al bus.

Sin embargo, este método tiene una gran desventaja y es que en caso de fallo en una estación se anularían todos los mensajes llegando a bloquearse el sistema. Para ellos hay que distinguir los errores esporádicos de los errores permanentes y los fallos locales de las estaciones. Esto se hace mediante una evaluación estadística de situaciones de error con el fin de reconocer los errores propios de un nodo y, a poder ser, entrando en funcionamiento de una manera que no afecta al resto de nodos.

5. Protocolo de comunicación Ethernet

En el presente capítulo, tal y como se comentó en el 4, se presenta una introducción a los principios básicos sobre el protocolo de comunicación Ethernet.

5.1. Introducción

El estándar Ethernet normalmente hace referencia al protocolo adjunto de comunicación, normalmente TCP/IP. Ethernet solo define la capa de enlace y las capas física del modelo OSI mientras que TCP/IP define las capas de transporte y de red respectivamente del mismo modelo.

Ilustración 30: Capas del modelo OSI. Obtenido de [39]

5.2. Tramas Ethernet

Los dos tipos de trama Ethernet que se utilizan en la industria son similares. La trama DIX V2.0, frecuentemente conocida como trama Ethernet II, consiste en

un preámbulo, 6 bytes para las direcciones de fuente y destino, un campo de 2 bytes usado para identificar protocolos de capas superiores, una variable de un byte seguido de una trama de 4 bytes que se utiliza como checksum. El IEEE 802.3 [40] divide el preámbulo en preámbulos de 7 bytes seguidos de un identificador de inicio de trama de 1 byte. El campo de datos actual incluye un 802.2 enlace de control lógico (LLC) que precede al campo de datos actual. El FCS permanece exactamente igual.

5.2.1. Preámbulo

El preámbulo DIX consiste en 64 bits alternando 1 y 0 pero terminando siempre en dos 1 para indicar que va a comenzar una trama válida. Esto crea una señal de 10 Mhz que sincronice los receptores de la red antes de que los datos lleguen. Ethernet usa la codificación Manchester.

El IEEE redefine el preámbulo para tener 7 bytes de preámbulo, el mismo que el preámbulo DIX, seguido de un bit de señalización de comienzo de trama que se parece al último bit del preámbulo DIX. No hay cambios en la operación entre el preámbulo DIX y el preámbulo IEEE y el byte SFD. Ambos preámbulos no son considerados parte de la trama cuando calculamos el tamaño de la trama.

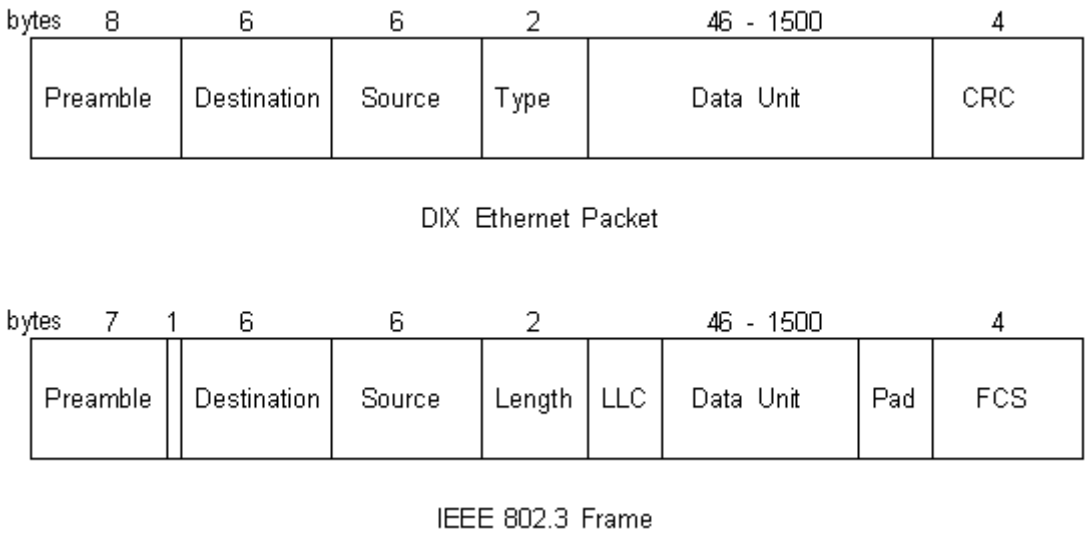

Ilustración 31: Tramas de Ethernet. Obtenido de [41]

5.2.2. Dirección de Destino

En el estándar DIC el primero de los 48-bit de la dirección de destino indican si la dirección es una dirección multicast o una dirección física. Un 0 indica una transmisión a la dirección de destino indicada mientras que 1 indica multicast o una dirección grupal.

El estándar IEEE define con mayor precisión el segundo bit de los 48-bits de destino para indicar si la dirección está administrada localmente o globalmente administrada. Este bit es 0 cuando la dirección está administrada globalmente; y está asignada por el proveedor de la interfaz Ethernet.

Una dirección de 48-bits donde todos ellos son 1 es una dirección broadcast tanto en los formatos DIX y IEEE indicando que la transmisión se dirige a todos los dispositivos de la red.

5.2.3. Dirección de la Fuente

La dirección de la fuente de emisión del mensaje se adjunta a la transmisión como una ayuda para los protocolos de capas superiores. No se utiliza para el control de acceso al medio. Para evitar el duplicado de ID de nodos para las direcciones globales, el adaptador Ethernet obtiene un identificador único llamado OUI. El OUI ocupa 24 bits y es la porción más significante de las direcciones de 48 bits. El fabricante asigna números secuenciales a cada tarjeta de red creando una dirección única a nivel mundial.

5.2.4. Campos de Tipo y Longitud

La intención original de Ethernet era que no se usara nunca la capa de enlace de datos como medio para proveer garantía de la entrega de los datos. Siempre se intentó que un protocolo de capas superiores se encargara de este servicio. Entonces solo era necesario identificar con un número que protocolo de capas superiores se está usando a través del campo de 2 bytes de la trama DIX.

El estándar 802.3 no incluye el campo “tipo”, pero a su vez define el campo de “longitud”. Un valor en este campo de 1518 o menos indica la longitud del campo mientras que un valor superior es ignorado, descartado o usado para funciones

privadas. Los valores fuera de los límites pueden usarse para identificar protocolos de capas superiores como en las tramas DIX.

Lo que es importante aquí es que las tramas DIX como IEEE son idénticos en términos de número de bits y longitud de los campos, ambas tramas pueden coexistir en la misma red, pero no son capaces de comunicarse uno con el otro. La mayoría del software existente TCP/IP usa tramas DIX.

5.2.5. Campo de Datos

Una trama Ethernet (sin encapsular por un protocolo) puede medir hasta 1500 bytes, pero no menos de 46 bytes. Esto es la trama DIX.

Aunque la longitud total disponible del campo de datos IEEE es el mismo que la trama DIX, la cabecera LLC reduce la cantidad de campo disponible para los datos o carga útil a la que normalmente está referida. Si la cabecera LLC y la carga útil son menos de 46 bytes, el campo de datos debe ser rellenado por 46 bytes para asegurar que la transmisión no se interprete como un paquete enano o un fragmento de paquete.

5.2.6. Secuencia de Verificación de Trama

Tanto el estándar DIX como el estándar IEEE usan 4 bytes para mantener el chequeo CRC-32 [42] en toda la trama desde la dirección de destino hasta el final del campo de datos. La estación que recibe la trama calcula su propio CRC-32, chequea los datos recibidos y compara los resultados con los calores CRC-32 transmitidos para verificar si coinciden puesto que esto indicaría una recepción correcta. Notar que no existe un mecanismo inherente en el protocolo de la capa de enlace de datos de Ethernet para informar al nodo fuente de que la recepción fue aceptada o rechazada debido a un chequeo CRC-32 fallido. Esta tarea se deja a cargo de algún protocolo de capas superiores.

5.3. Medio Físico

Aunque Ethernet fue diseñado originalmente como un sistema de bus coaxial, medios físicos sustitutos han evolucionado desde el comienzo de los 80. El comité IEEE 802 ha definido varios medios físicos [43].

5.3.1. 10BASE5

El Ethernet originalmente se configuró como un sistema de buses con un cable coaxial delgado como medio. Esto es lo que se especificó en el estándar DIX de 1980. Un transceptor externo llamado unidad de adhesión al medio (MAU) se agarran a ciertos puntos particulares del cable marcados por rayas cada 2.5 metros. Se conecta un cable desde el puerto de la unidad de adhesión a la interfaz (AUI) al adaptador de red Ethernet. El puerto del AUI es un DB-15 [44].

El cable coaxial puede ser de hasta 500 metros y el cable que conecta el AUI con la tarjeta de red está limitado a una longitud máxima de 50 metros. Un total de 100 transceptores pueden situarse en una red troncal. Se pueden unir en cascada distintas redes troncales usando repetidores hasta 2000 metros.

En 1985 el IEEE estandarizó esta configuración como 10BASE5 que significa que implemente un ancho de banda de 10 Mbps a una longitud máxima de 500 metros.

Sin embargo, los cables coaxiales delgados son pesados y voluminosas y esta topología no es siempre la recomendada para cablear en una planta. Resolver los problemas en un segmento con 100 estaciones es una pesadilla con lo que en la actualidad no se utiliza esta topología. Además, este cable no soporta la tecnología Fast Ethernet.

5.3.2. 10BASE2

La solución a los problemas que acarrea el sistema 10BASE5 fue el uso de Thinnet [45] o Cheapernet [46] estandarizados en 1985 como 10BASE2.

Thinnet es una topología de bus con transceptores internos. Un cable RG-58/u [47] interconecta hasta 30 estaciones distanciadas como máximo 185 metros. Los segmentos se pueden repetir hasta alcanzar una longitud de 740 metros.

Aunque es más fácil de instalar que 10BASE5, las nuevas instalaciones prefieren la utilización de cables de par trenzado ya que este cable no soporta el protocolo Fast Ethernet.

5.3.3. 10BASE-T

En 1990, el IEEE publicó 10BASE-T después de un trabajo pionero que introdujo los cables de par tranzado y la topología en estrella a las instalaciones Ethernet [48].

Los adaptadores de red de este medio físico tienen transceptores internos y conectores RJ-45. Normalmente dos pares de cables sin protección se anexan a un hub porque las conexiones de bus no están permitidas. La conexión entre un adaptador y el hub no puede exceder los 100 metros de longitud. La longitud de las conexiones hub a hub puede variar dependiendo del medio usado. Si se utiliza un cableado Thinnet la longitud máxima es 185 metros y con cable coaxial tenemos hasta 500 metros.

La topología en estrella resulta más fácil a la hora de resolver los errores que una topología en bus, sin embargo, la seguridad del hub debe ahora considerarse en la confianza general del sistema. Otra razón para que la tecnología se centre en los pares trenzados es que la tecnología de Fast Ethernet se basa en estos.

5.3.4. 10BASE-F

El estándar 10BASE-F es en realidad una serie de estándares de fibra óptica. La fibra óptica proporciona largas distancias, altas velocidades, inmunidad ante el ruido y aislamiento eléctrico. Existen tres estándares:

- 10BASE-FL: Este estándar sustituye al estándar FOIRL[8].
- 10BASE-FB: No es popular.
- 10BASE-FP: Utiliza una tecnología con hub pasivos nada comunes.

El estándar 10BASE-F requiere fibra óptica 62.5/125µm para cada enlace. Es posible mantener distancias de transmisión de hasta 2km, así como operaciones full-duplex.

[8] Fiber Optic Inter Repeater Link. FOIRL es un estándar generalmente usado para extender la longitud de la rama por encima de 328 pies. Sin embargo, está limitado a 0.6 millas por segmento.

5.4. Control de Acceso al Medio

Cuando una estación quiere transmitir, primero espera a una ausencia de portadora que indicaría que otra estación está transmitiendo. Tan pronto como se detecte silencio, la estación que estaba esperando a transmitir continúa aplazando la transmisión hasta que el tiempo IFG (Interframe Gap*)* haya expirado, lo cual es como mínimo 96 tiempos de bits, es decir, unos 9.6µs.

Si la portadora sigue ausente, la estación comienza a transmitir mientras observa si se produce alguna colisión. Si no se detecta colisión alguna, la estación que se encuentra transmitiendo asume que se ha realizado correctamente. Si se detecta alguna colisión temprana, por ejemplo, una que ocurra durante el preámbulo, la estación continúa enviando el preámbulo más 32 bits de datos llamados señal jam*.* Esto asegura que otras estaciones también noten las colisiones.

Después de la colisión, la estación de transmisión dará back off la retransmisión utilizando el algoritmo de back off. Si no se detectan colisiones después de 512 tiempos de bit (sin contar el preámbulo), la estación asume que ha adquirido el canal y no ocurrirán colisiones posteriores en la red de trabajo. El contador de colisiones de resetea. Este tiempo de 512 bits (51.2 µs) se llama “ranura de tiempo” y es crítica en la forma que Ethernet arbitra el acceso al cable.

5.4.1. Dominio de Colisión

Esta ranura de tiempo define el límite superior del retraso de propagación total de un símbolo transmitido desde el comienzo de la red hasta el punto más lejano y regreso. Esto incluye el tiempo que tarda el símbolo en viajar a través del cable, repetidores y MAUs. Sin embargo, independientemente del camino, el retraso en la propagación resultante debe ser menor que la ranura de tiempo. Por lo tanto, la ranura de tiempo define el diámetro máximo de la red Ethernet el cual limita el dominio de colisión. Un dominio de la colisión que exceda el diámetro máximo de la red viola el mecanismo de control de acceso al medio resultando en una operación no fidedigna.

Las colisiones pueden generar paquetes runt que presentan menos de 512 bits de longitud, los cuales se pueden detectar por los nodos que tienen que recibir

el paquete y descartarlo de manera acorde. Esto explica el porqué es importante que un mínimo de tramas válidas Ethernet siempre se envía para distinguir paquetes válidos de fragmentos de paquetes. Un mínimo de 46 bytes en el campo de datos asegura que la trama de Ethernet válida sea de 512 bits. Los mensajes de control son típicamente cortos así que se tiene que recordar que la trama Ethernet más corta es de 64 bytes.

Si el diámetro de la red es pequeño, la detección de colisión es rápida y los resultantes fragmentos de colisión son pequeños. Conforme el diámetro de la red se va incrementando se pierde más tiempo en detectar las colisiones y los fragmentos se vuelven más largos.

Aumentar el diámetro de la red agrava el problema de las colisiones. El silencio en la línea no necesariamente significa que un transmisor distante no ha enviado un paquete a través del cable, que eventualmente resulta en colisión.

5.4.2. Detección de colisiones

Una colisión se define como dos estaciones tratando de transmitir al mismo tiempo. En los transceptores de cable coaxial, no hay una circuitería para detectar el nivel de continua de la señal en el cable. Este es un indicador de colisión. En la fibra óptica y en los interfaces de par trenzado con circuitería separada de recepción y transmisión, las colisiones se detectan por simultáneas recepciones y transmisiones de datos.

Solo los transmisores miran las colisiones y es su responsabilidad reforzar una colisión con la señal jam. Los receptores solo miran los paquetes válidos y automáticamente descartan los paquetes que se han creado debido a las colisiones. Una vez que se detecta una colisión por simultáneos transmisores, estos transmisores seguirán el algoritmo de back off.

5.4.3. Algoritmo de Back off

Cuando ocurre una colisión en la red, los transmisores que estén colisionando darán back off la retransmisión por un tiempo determinado utilizando un algoritmo de back off. Este algoritmo requiere que cada transmisor espero un numero de

ranuras de tiempo (51.2 μs) antes de intentar una nueva secuencia de transmisión. El número de ranuras de tiempo viene determinado por la ecuación:

$$0 < r < 2^k \qquad donde\ k = \min(n, 10)$$

La variable k es el número de colisiones limitadas a un máximo de 10, Entonces, r se sitúa en el rango entre 0 y 1023. El valor de r se determina por un proceso aleatorio con cada nodo Ethernet. Conforme aumente el número de colisiones consecutivas, el rango de los posibles tiempos de back off incrementa exponencialmente. El número de los posibles reintentos están limitados a 16.

Un gran número de reintentos indica una red ocupada con más estaciones a la espera de transmitir que las originalmente diseñadas. Esto es el motivo por el que el rango de tiempos de back off se incrementa exponencialmente para proveer más ranuras de tiempo para posibles estaciones adicionales. Con 10 reintentos, se asume que existen 1024 transmisores simultáneos.

Esto se convierte en el límite superior de estaciones que pueden coexistir en una red Ethernet. En realidad, esto es un límite lógico ya que es físicamente imposible tener varias estaciones en un dominio de colisiones sin violar las reglas de cableado.

NÚMERO DE INTENTOS	NUMERO DE ESTACIONES	NÚMEROS ALEATORIOS	TIEMPOS DE BACKOFF
1	1	0 – 1	0 – 51.2 µs
2	3	0 – 3	0 – 153.6 µs
3	7	0 - 7	0 – 358.4 µs
4	15	0 - 15	0 – 768 µs
5	31	0 - 31	0 – 1587.2 µs
6	63	0 - 63	0 – 3225.6 µs
7	127	0 - 127	0 – 6502.4 µs
8	255	0 - 255	0 – 13056 µs
9	511	0 - 511	0 – 26163.2 µs
10	1023	0 - 1023	0 – 52377.6 µs
11	1023	0 - 1023	0 – 52377.6 µs
12	1023	0 - 1023	0 – 52377.6 µs
13	1023	0 - 1023	0 – 52377.6 µs
14	1023	0 - 1023	0 – 52377.6 µs
15	1023	0 - 1023	0 – 52377.6 µs
16	Demasiadas	N/A	Trama Descartada

Tabla 10: Rango de Backoff en función de las colisiones. Fuente Propia

6. Diseño de la solución

Para la realización de este proyecto necesitamos utilizar materiales tanto físicos como software. En los siguientes apartados, vamos a describir los materiales que se van a utilizar en la implementación, así como los cálculos matemáticos y electrónicos necesarios para el correcto diseño del proyecto.

Con estas descripciones y junto al apartado 1.2 estaremos definiendo las especificaciones que debe cumplir el proyecto tras finalizar su implementación.

6.1. Hardware utilizado

Para la realización de este proyecto se necesitan 3 categorías de hardware. Por un lado, tenemos los sensores de los cuales obtendremos valores analógicos y digitales, por otro lado, tenemos los shield que nos permiten utilizar ciertos pines para realizar una comunicación CAN Bus y, por último, y más importante, tenemos los dispositivos SoC.

6.1.1. Dispositivo SoC

Tras las conclusiones obtenidas en el aparato **¡Error! No se encuentra el origen de la referencia.** se ha decidido utilizar para este proyecto la placa Intel Galileo.

El motivo de esta decisión, además de por sus características técnicas, que ya hemos comentado en detalle en el apartado **¡Error! No se encuentra el origen de la referencia.**, es el hecho de que al presentar una mayor capacidad de procesamiento, puesto que podemos instalarle un sistema operativo [49], nos abre nuevas líneas de investigación posteriores, así como un mayor atractivo comercial y empresarial.

A nivel de este proyecto, es la única placa de la comparativa que además de disponer de pines de conexionado de señales permite las comunicaciones CAN Bus (añadiendo un shield) y Ethernet de forma nativa.

A nivel educativo también es más interesante que el resto de dispositivos comparados en el apartado 3.4 debido a que la gran mayoría de proyectos

académicos y DIY[9] open hardware están realizados con Arduino y/o Raspberry Pi, con lo cual estamos con este TFM indirectamente ofreciendo información sobre la funcionalidad práctica de las placas Intel Galileo.

6.1.2. CAN Shield

Para poder utilizar los pines de SPI para la comunicación CAN BUS necesitamos un shield que nos lo permita.

Uno de los más conocidos comercialmente es el CAN Shield de SeeedStudio [50].

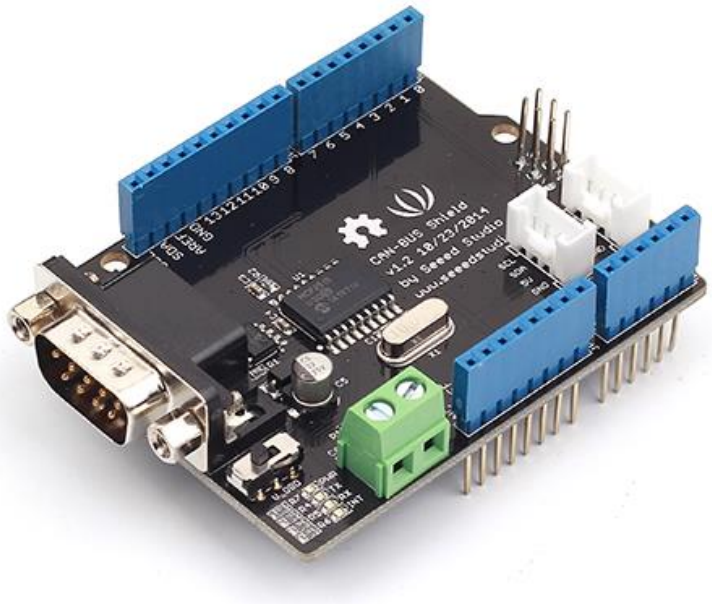

Ilustración 32: CAN Bus Shield de SeeedStudio. Obtenida de [50]

Pese a que está pensado para ser usado única y exclusivamente con Arduino o Seeeduino, nuestra placa Intel Galileo es compatible con este shield debido a que la situación de los pines la hace compatible con todos los shields pensados para Arduino Uno.

Este shield está formado básicamente por un controlador de CAN Bus con interfaz SPI MCP2515 [51] junto con un transceptor MCP2551 [52] y un conversor OBD-II [53]. Gracias a este conversor podemos conectar las dos

[9] Do It Yourself:

placas mediante DB9 o con los cables directos respetando la polaridad de CAN_H y CAN_L tal y como se realizan las conexiones a bordo de los buques.

Otra de las ventajas de este shield frente a otros del mercado es que nos permite tener libres todos los pines analógicos y 8 pines digitales de nuestra placa Intel Galileo con lo cual el nodo esclavo puede estar conectado a los sensores que debe medir y a este shield a la vez sin problemas.

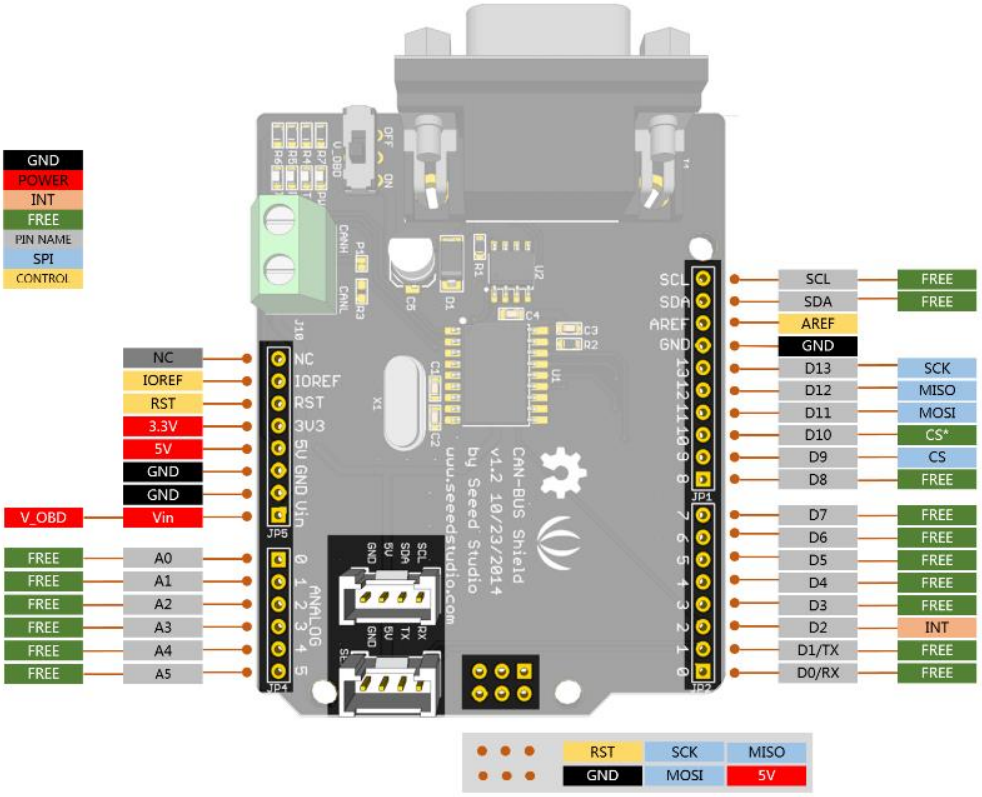

Ilustración 33: Mapeo de los pines de CAN Bus Shield de SeeedStudio. Adaptada de [54]

No podemos finalizar este apartado sin mencionar todas sus características:

- Implementa CAN BUS 2.0B con lo cual podemos alcanzar velocidades de hasta 1Mb/s.
- La velocidad de la interfaz SPI es de hasta 10MHz.
- Permite tanto tramas de CAN Bus estándar como extendidas.
- Presenta dos buffers de recepción con almacenamiento de mensajes priorizados.
- Conector industrial DB9
- LED´s de indicación de estado.

6.1.3. Sensores

En lugar de simular los datos a procesar por el sistema, los vamos a obtener de sensores reales.

Para tener un sistema lo más completo posible vamos a utilizar dos sensores (uno de temperatura para las señales analógicas y otro de humedad-temperatura para las señales digitales.

Sensor de Temperatura

El sensor de temperatura utilizado es un sensor LM35DZ [55].

Ilustración 34: Sensor LM35DZ. Obtenida de [56]

El sensor LM35 de Texas Instrument nos proporciona una tensión de salida de 10mV por cada ºC de temperatura, por tanto, su salida es directamente proporcional a la medida realizada. Su medición abarca, este modelo en concreto, entre 0 y 100ºC utilizando un rango de alimentación de entre 4 y 30V.

Como se ve en la Ilustración 34 presenta un encapsulado de tipo TO-92[10], presentando la siguiente configuración de pines:

[10] Transitor Outline Package, Case Style 92

Ilustración 35: Configuración pines LM35DZ. Adaptada de [57]

El patillaje del LM35 que muestra la Ilustración 35 nos informa que los pines externos son para alimentación, mientras que el canal central proporciona la medición en una referencia de tensión a razón de, como hemos comentado en los párrafos anteriores, 10mv/ºC.

Por tanto, el esquema eléctrico que necesitamos es el siguiente:

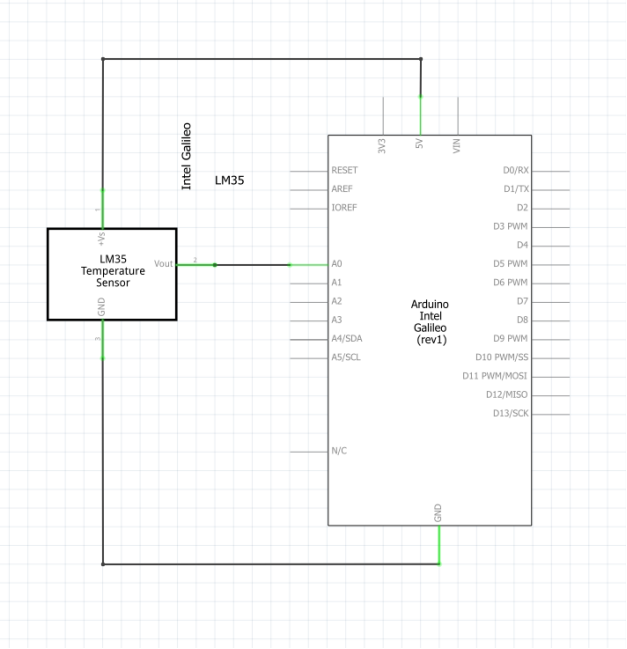

Ilustración 36: Esquema eléctrico conexionado LM35. Fuente Propia[11]

[11] Para la realización de los esquemas eléctricos y de conexionado se ha utilizado el software libre Fritzing [58].

Mientras que el montaje en la protoboard sería el siguiente:

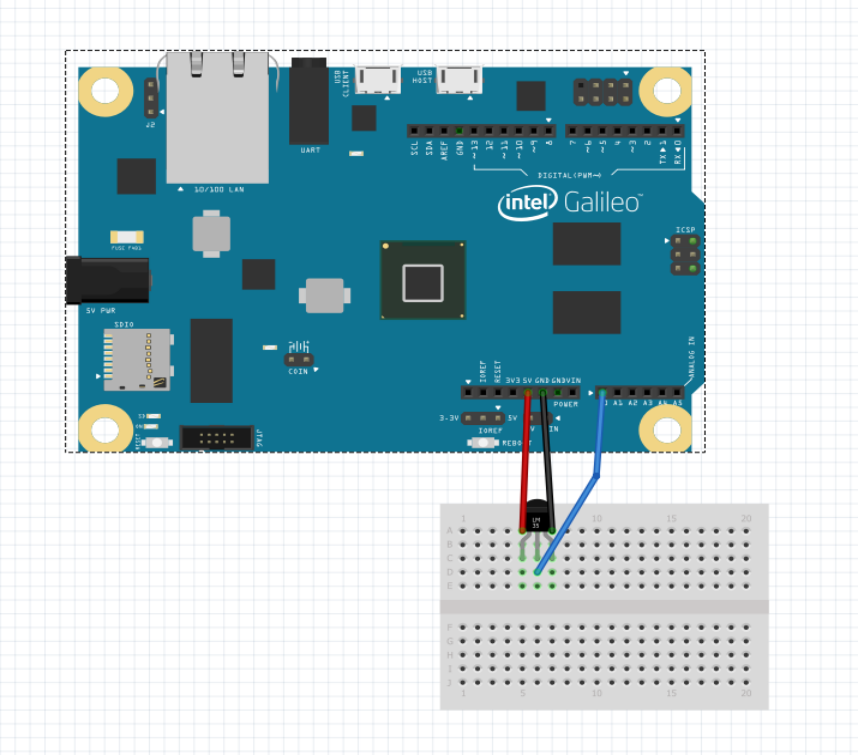

Ilustración 37: Montaje en protoboard LM35. Fuente Propia.

Sensor de humedad

El sensor de humedad utilizado es el modelo DHT11 fabricado por Adafruit. Este sensor también nos proporciona también la temperatura, así que lo usaremos para comparar este valor con el obtenido con el sensor LM35.

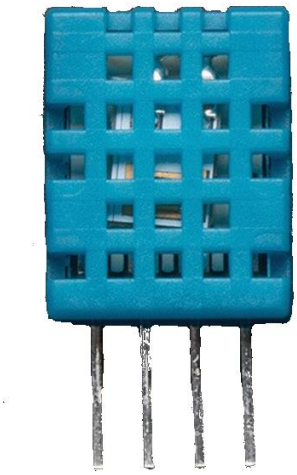

Ilustración 38: Sensor DHT11. Fuente Propia

El sensor DHT11 es un sensor de humedad y de temperatura basado en un sensor de humedad capacitivo un termistor. A pesar de esto, su salida es una señal digital, no analógica, [59] con lo cual hay que tener cuidado con el tiempo de lectura ya que este sensor solo nos ofrece un nuevo valor cada 2 segundos.

Sus características técnicas son:

- Tensión de alimentación de entre 3 y 5V.
- Máxima corriente de 2.5mA durante la conversión.
- Rango de medición de humedad de 20-80% con una precisión del 5%.
- Rango de medición de temperatura de 0-50ºC con una precisión de ±2ºC.
- Frecuencia de muestreo máxima de 1Hz.

La configuración de sus pines es bastante especial ya que pese a tener 4 pines, solo se deben conectar 3, pasando a tratarse como el encapsulado del apartado anterior.

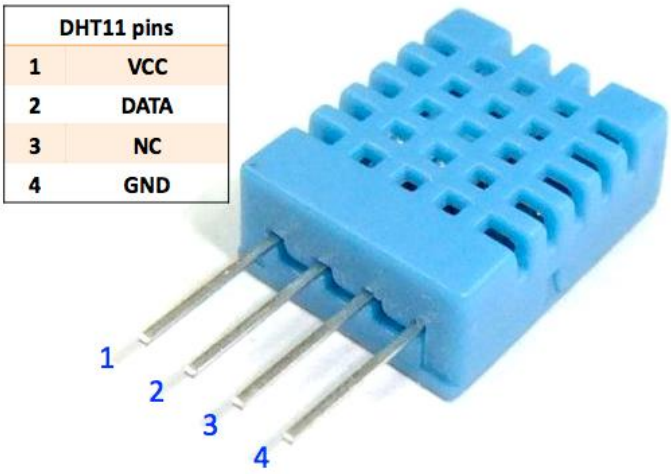

Ilustración 39: Configuración de pines DHT11. Obtenido de [60]

El patillaje del DHT11 de la Ilustración 39 nos informa que los pines externos son alimentación, que el segundo pin (empezando por la izquierda) es la salida y el tercer pin no se conecta a nada.

Por tanto, el esquema eléctrico que necesitamos para este componente es el siguiente:

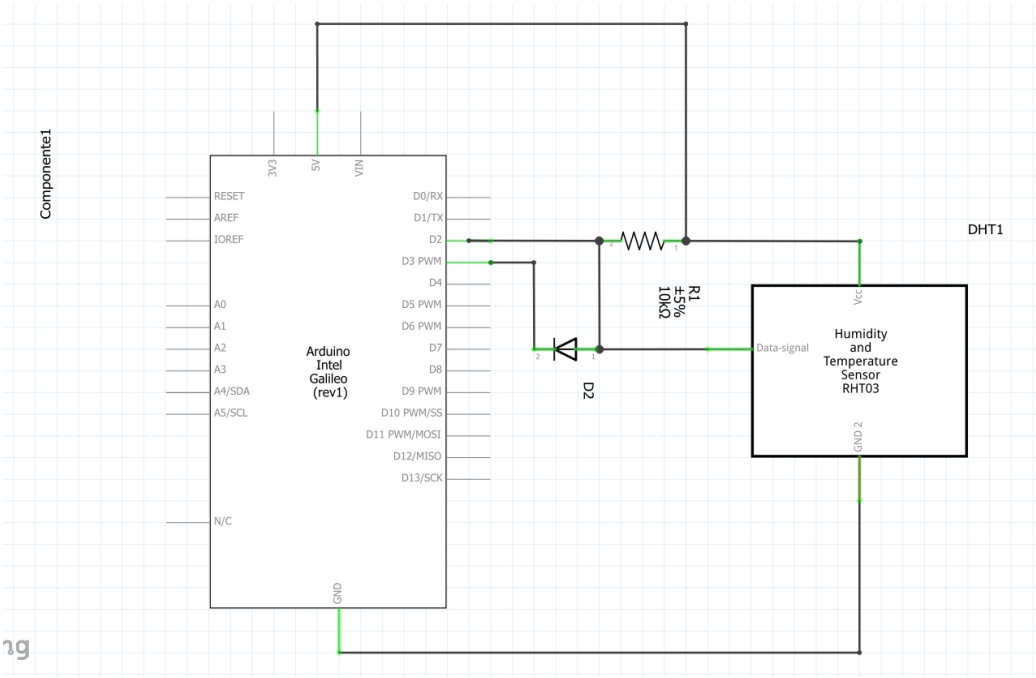

Ilustración 40: Esquema eléctrico conexionado DHT11. Fuente propia

Mientras que el montaje en la protoboard, para el esquema de la Ilustración 40 sería el siguiente:

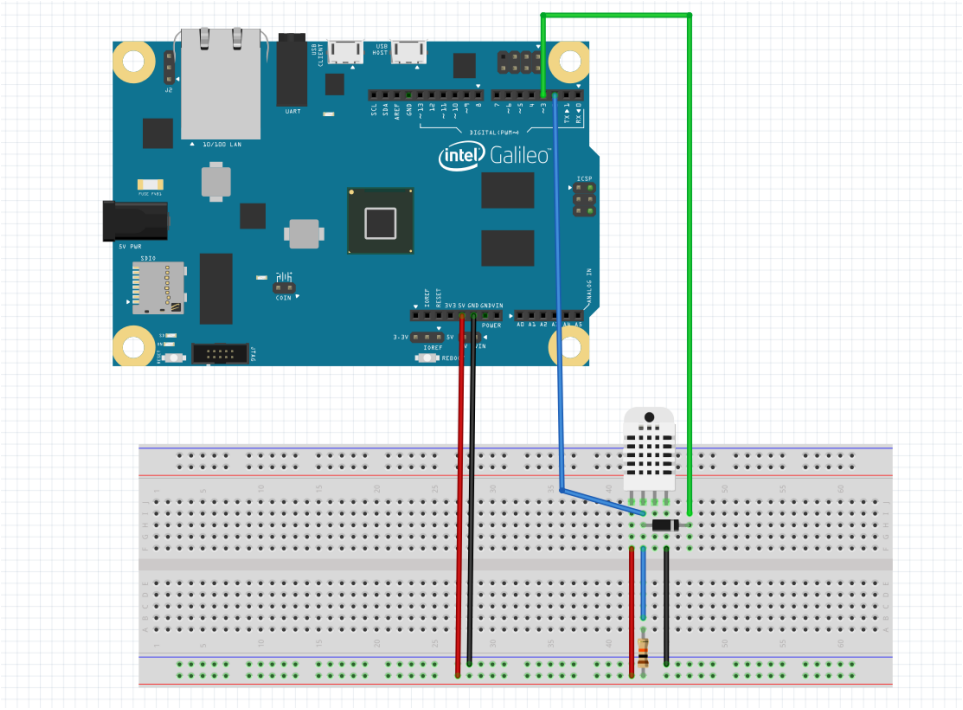

Ilustración 41: Montaje en protoboard DHT11. Fuente propia

Como se observa tanto en Ilustración 40 e Ilustración 41 se ha utilizado un diodo rectificador entre el pin de 2 del sensor de humedad DHT11 y el pin 3 de la placa Galileo.

Se ha decidido realizar este cambio debido a que Galileo usa expansores IO para los GPIO que controlan la dirección del pin (recordemos que sus pines digitales son de entrada/salida como vimos en 3.4.2) pero, estos expansores están conectados a Galileo vía I2C lo que hace que cambiar la dirección del pin requiera mucho tiempo.

La mejor manera de hacerlo es convertir el dispositivo de un hilo en uno de dos hilos: uno dedicado a la entrada y otro a la salida.

6.2. Software

En este apartado vamos a explicar los programas básicos que son necesarios para realizar la implementación del proyecto.

6.2.1. Arduino IDE

Galileo se programa utilizando el mismo IDE de Arduino aunque cambiando el sistema operativo nativo, Yocto [28] por una imagen de Linux podemos programar nuestra Galileo utilizando Phyton y Node.js así como utilizar SSH.

El IDE[12] de Arduino es un pequeño programa que se instala en el ordenador y con un entorno muy sencillo nos permite escribir los programas que debe ejecutar nuestra placa Arduino. Es multiplataforma, con lo cual lo podemos instalar bajo Windows, Linux y Mac. En nuestro caso se instalará bajo Windows 10.

Este programa se puede descargar desde la propia web de Arduino ya que es software libre.[13]

[12] Integrated Development Environment

[13] La diferencia entre software libre y código abierto es que, aunque ambos son gratuitos el código del programa open software pertenece a una empresa privada y los usuarios no lo pueden modificar. [61]

También existe otro entorno de desarrollo compatible con Intel Galileo llamado Intel XDK [62] que permite desarrollar aplicaciones móviles, desarrollo de software basado en Node.js y aplicaciones IoT.

En nuestro caso al trabajar con Windows 10, y dadas las características del proyecto, vamos a descargar la aplicación de Arduino IDE [63] para este sistema operativo desde la store.

Una vez ejecutado, nos encontramos directamente con el entorno de desarrollo.

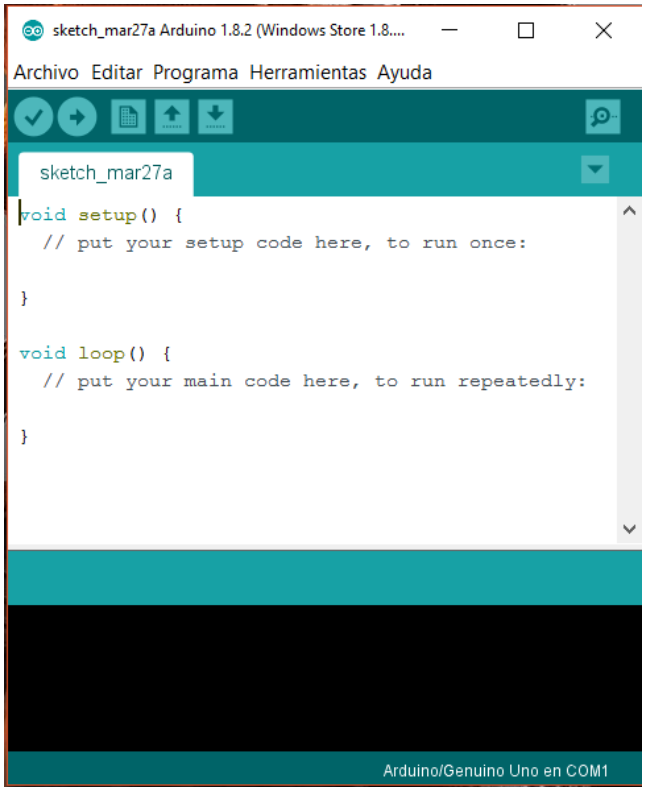

Ilustración 42: Entorno de desarrollo Arduino IDE. Fuente propia

Si nos fijamos en la barra de trabajo tenemos las siguientes opciones (de izquierda a derecha):

Ilustración 43: Barra de herramientas de trabajo Arduino IDE. Fuente Propia

- Compilar el sketch.
- Programar el sketch en la placa.

- Crear un nuevo sketch.
- Cargar un sketch.
- Guardar el sketch.

Si pulsamos en Herramientas->Board podemos la elegir el tipo de placa Arduino sobre la que queremos trabajar. Si nos damos cuenta en este entorno de desarrollo de manera nativa no aparece la placa Intel Galileo.

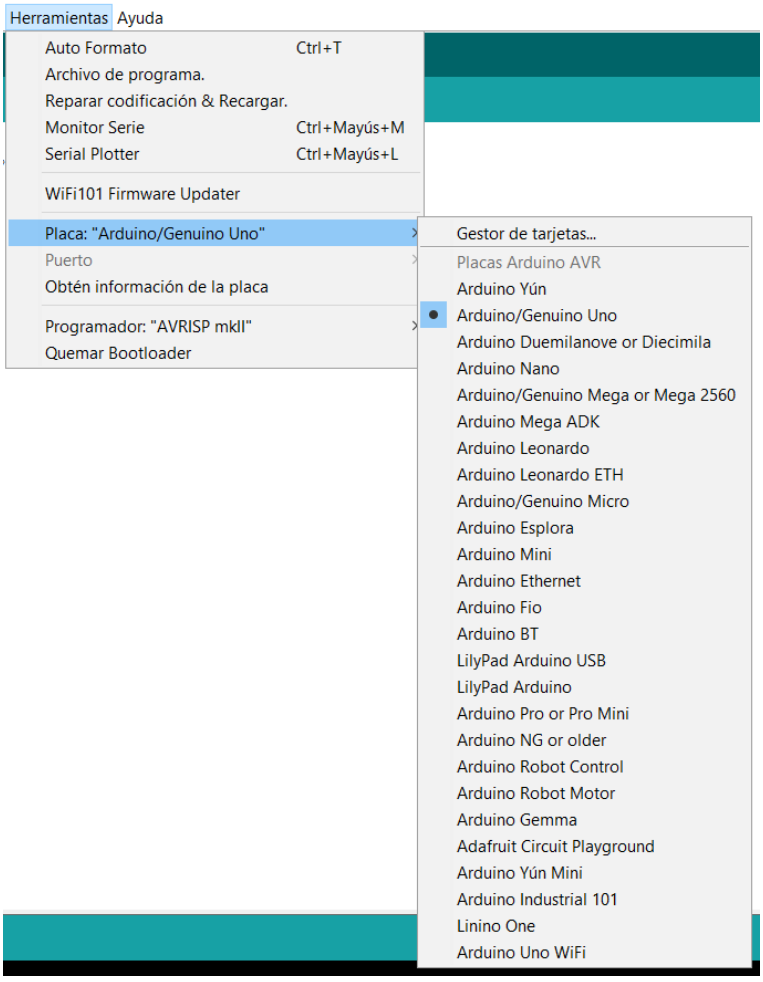

Ilustración 44: Placas compatibles con Arduino IDE. Fuente propia

Como vemos en la Ilustración 44, tenemos que configurar el entorno de desarrollo para hacerlo compatible con la placa Intel Galileo. Para ello, pulsamos en Gestor de tarjetas.

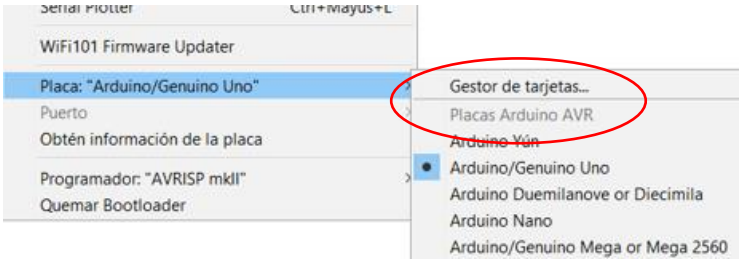

Ilustración 45: Selección Gestor de tarjetas Arduino IDE. Fuente propia

Una vez abierto nos encontramos un desplegable con todos los paquetes extra que podemos instalar para poder trabajar sobre más placas.

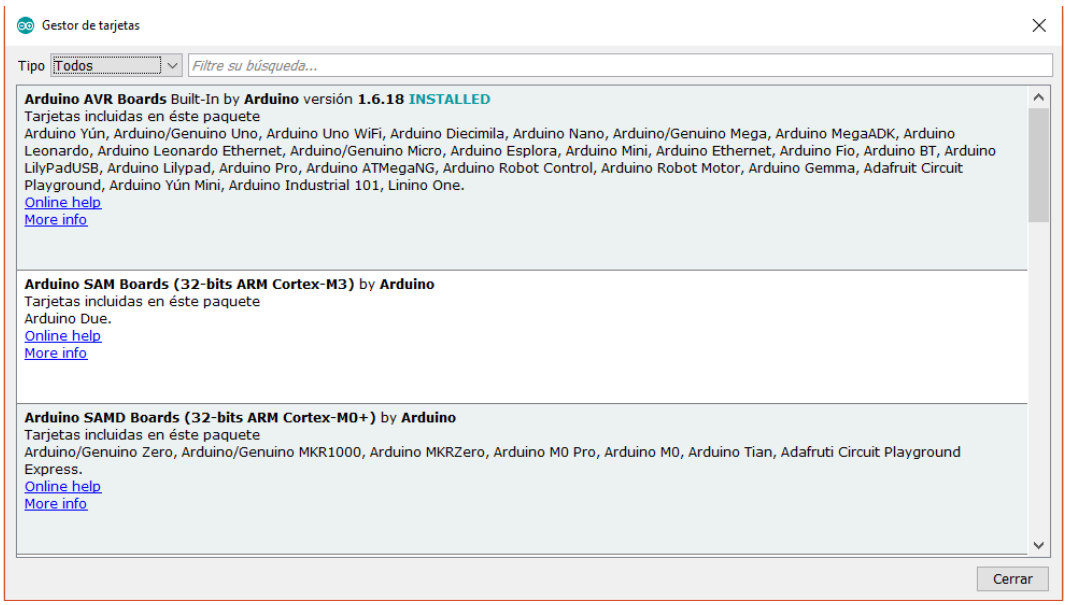

Ilustración 46: Gestor de tarjetas Arduino IDE. Fuente propia

Si pulsamos en el desplegable de tipo y seleccionamos Arduino Certified nos aparecerá el paquete Intel i586 que corresponde a la tarjeta Galileo.

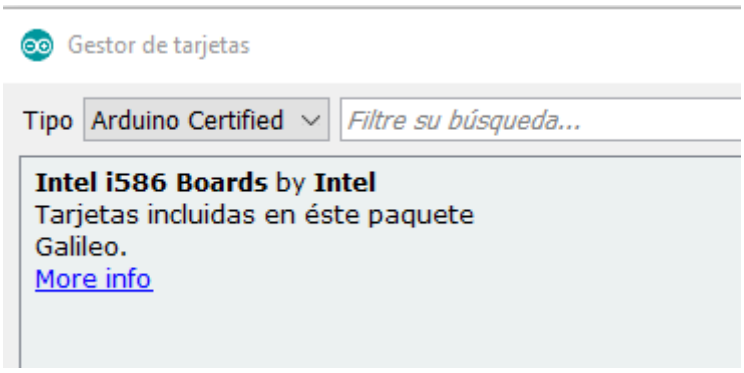

Ilustración 47: Gestor de tarjetas: Arduino Certified. Fuente propia

Una vez instalado ya tenemos el software listo para implementar este proyecto.

6.2.2. Drivers placa Intel Galileo

Para que la placa Intel Galileo funcione sin problemas con el entorno de desarrollo Arduino que se ha detallado en el apartado 6.2.1 debemos seguir una serie de pasos especiales.

Tras conectarlo a la luz eléctrica, cuando el led serigrafriado como USB se encienda lo conectaremos vía USB a un ordenador. El ordenador, lo detectará como un puerto COM genérico.

Desde la web de descargas de Intel [64] nos descargamos el archivo IntelGalileoFirmwareUpdater en formato .rar correspondiente a nuestro sistema operativo.

Al extraerlo veremos que incluye un ejecutable y una carpeta llamada Galileo Drivers.

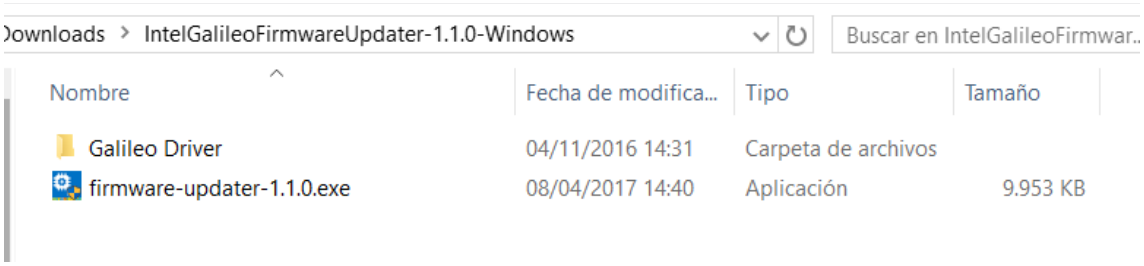

Ilustración 48: Contenido del archivo IntelGalileoFirmwareUpdater. Fuente propia

Desde el administrador de dispositivos, botón derecho sobre el puerto COM que nos ha aparecido y seleccionamos la opción de actualizar controlador de dispositivo.

Aquí elegiremos la ruta donde hemos extraído la carpeta Galileo Driver de la Ilustración 48. Tras esta acción veremos como el nombre del puerto COM ha cambiado a Galileo.

Aunque en muchas fuentes se habla solo de la instalación de los drivers, otras recomiendan realizar una actualización del firmware [65] previamente a comenzar a trabajar.

Para esto solo tenemos que ejecutar el archivo .exe de la Ilustración 48 y tras unos minutos (el proceso tarda unos 4 minutos) en los que no se puede desconectar la placa Intel de la alimentación, queda la placa lista para comenzar con la implementación del capítulo 7.

6.3. Diseño final protocolo CAN BUS

Teniendo en cuenta los elementos descritos en los apartados anteriores. El diseño final que se implementará en el capítulo 7 para la utilización del protocolo CAN Bus consiste en dos nodos, un esclavo y otro maestro. El nodo esclavo, obtendrá los valores de temperatura y humedad de los sensores estudiados en el capítulo 6.1.3.

Tras procesar estos datos, los enviará utilizando el Can Shield que hemos visto en el capítulo 6.1.2 al nodo maestro, el cual, utilizando la misma versión de shield, leerá los datos.

Nosotros veremos si estos datos son los correctos a través del puerto serie del nodo maestro.

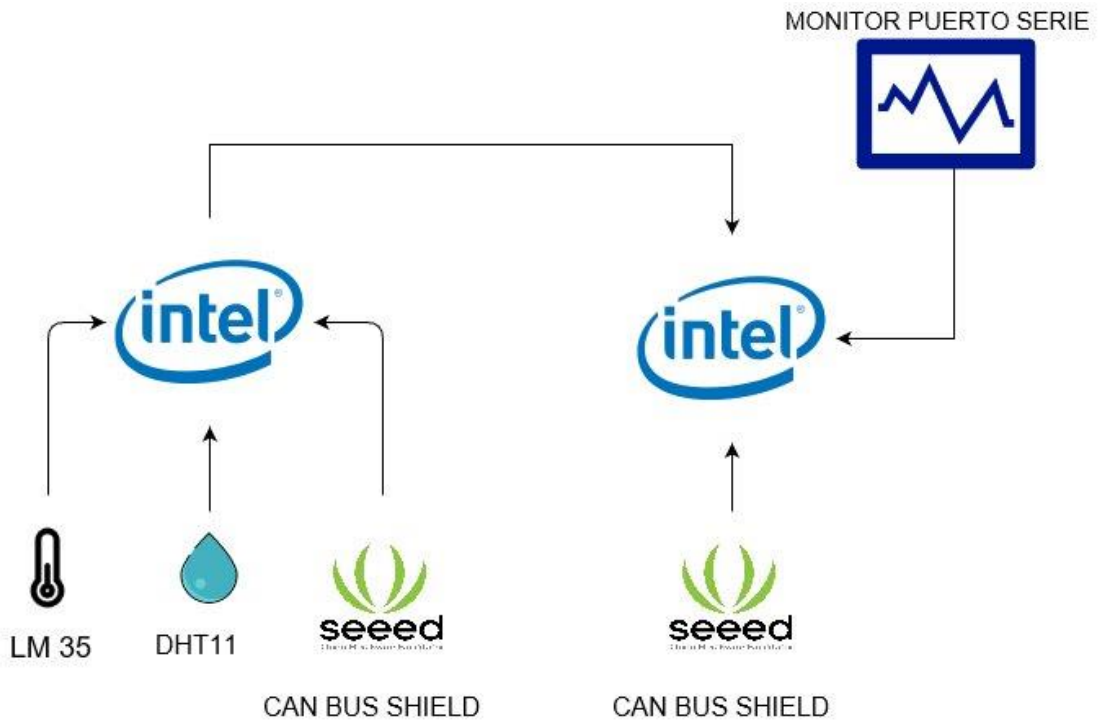

Ilustración 49: Diseño de la solución con CAN BUS. Fuente Propia

6.4. Diseño final protocolo Ethernet

El diseño final con el protocolo Ethernet es mucho más sencillo que el diseño con CAN-Bus estudiado en el apartado 6.3. Este diseño consistirá en la placa Intel Galileo que anteriormente hemos denominado “nodo esclavo” junto con los mismos sensores planteados en el apartado 6.1.3. Conectaremos esta placa a un router al cual conectaremos también un PC que solo utilizaremos para observar que se están enviando los datos correctamente a través de un navegador web.

Podríamos conectar la segunda Intel Galileo (que anteriormente hemos denominado “nodo maestro”) al router para que reciba los datos, pero resulta un ejercicio redundante que no añade complejidad al proyecto, cosa que si sucedía en el caso del diseño con CAN Bus del aparatado 6.3.

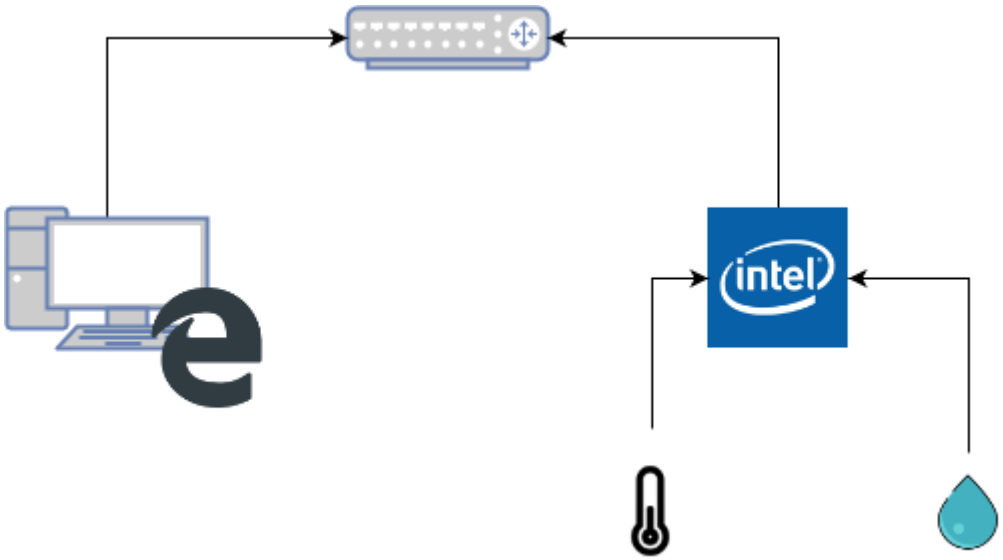

Ilustración 50: Diseño de la solución con Ethernet. Fuente Propia

7. Implementación de la solución

Vamos a dedicar este capítulo a implementar la solución utilizando los diseños y realizados y elementos explicados en el capítulo 6.

En lugar de realizar toda la implementación en un solo bloque, vamos a dividir ésta en cada uno de los entregables wbs para de tal modo poder depurar el código y analizarlo más cómodamente.

7.1. Nodo Esclavo

El nodo esclavo es aquella placa Intel Galileo que se va a encargar de adquirir los datos de los sensores, procesarlos y enviarlos por el CAN.

7.1.1. Lectura Sensor LM35

Para este entregable necesitamos realizar el siguiente conexionado:

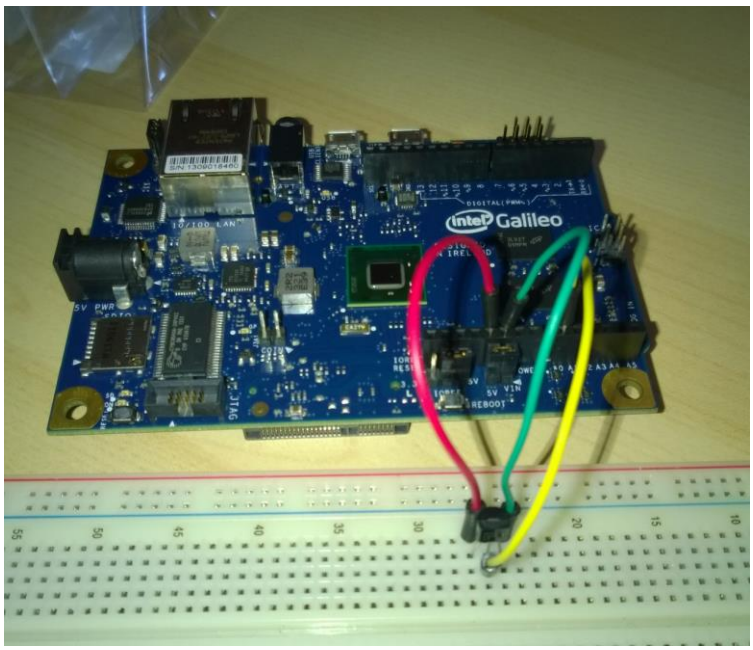

Ilustración 51: Montaje implementación lectura sensor LM35. Fuente Propia

El código utilizado es el siguiente:

```
const int LM35Pin = A0;
void setup() {
  Serial.begin(9600);
```

```
}

void loop() {
  int lecturaLM35 = analogRead(LM35Pin);
  float milivoltios = (lecturaLM35 / 1023.0) * 5000;
  float grados = milivoltios / 10;
  Serial.print(grados);
  Serial.println("ยบC");
  delay(1000);

}
```

No necesitamos ninguna librería externa ya que todo el proceso lo realiza la Intel Galileo.

En la primera línea definimos a qué pin analógico físico vamos a conectar la entrada de datos del sensor LM35. Después activamos el puerto serie para poder testear este sketch desde el monitor de puerto serie del Arduino IDE.

Ya dentro de la función loop, función que se va a ejecutar en bucle indefinidamente, leemos los datos del pin definido anteriormente, convertimos la señal y mostramos el valor obtenido en el puerto serie.

Si nos fijamos en la línea 9, lo que se está realizando aquí es la conversión digital a analógico del valor leído. Para ello primero lo dividimos entre el número de cuentas, en este caso 1023 ya que el conversor ADC que utiliza Intel Galileo es de 10 bits (0-1023) y lo multiplicamos por la tensión de referencia, en nuestro caso 5V, aunque podríamos también configurar el jumper de la placa para trabajar a 3.3V. Como queremos un valor en mili voltios multiplicamos por 1000 la señal obtenida (ya que hasta ahora tenemos voltios).

Después aplicamos el comportamiento del sensor LM35 que, como explicamos en 6.1.3, nos proporciona una sensibilidad de 10mV por cada grado de temperatura.

Si ejecutamos el monitor de línea serie de Arduino obtenemos lo siguiente:

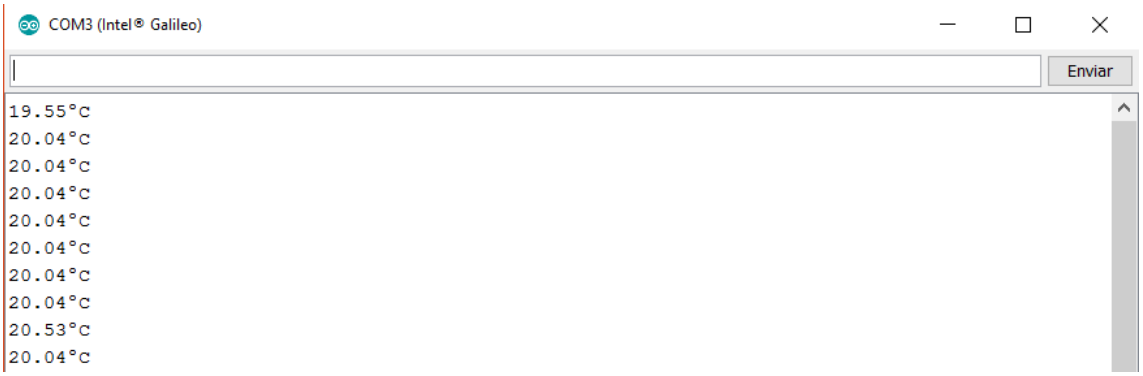

Ilustración 52: Monitorización de puerto serie de la ***¡Error! No se encuentra el origen de la referencia.****. Fuente Propia.*

7.1.2. Lectura Sensor DHT11

Para este apartado necesitamos realizar el siguiente conexionado:

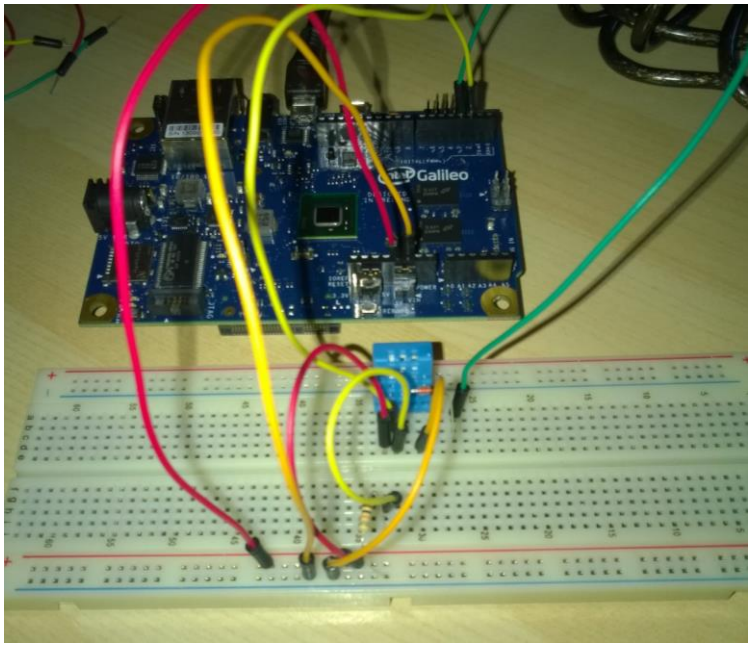

Ilustración 53: Montaje para la implementación de la lectura del Sensor DHT11. Fuente Propia

Para poder realizar la programación de la lectura del sensor necesitamos utilizar la librería que nos proporciona el fabricante Adafruit [66]. Sin embargo, para poder trabajar a 2 hilos con el conexionado de la Ilustración 41 hemos requerido modificar la librería tal y como se explica en [67].

Para instalar una librería solo hay que seguir los pasos que se explican en los tutoriales de la web de Arduino [68] (recordemos que estamos trabajando con el entorno de desarrollo de Arduino).

El código utilizado es el siguiente:

```
#include <DHT.h>

#define DHTIN 2
#define DHTOUT 3
#define DHTTYPE DHT11

DHT dht(DHTIN,DHTOUT,DHTTYPE);
void setup() {
  Serial.begin(9600);
  Serial.println("Lectura Sensor DHT11");
  dht.begin();

}

void loop() {
  delay(2000);

  float h=dht.readHumidity();
  float t=dht.readTemperature();

  if(isnan(h)||isnan(t)){
    Serial.println("Error de lectura");
    return;
  }

  Serial.print("Humedad: ");
  Serial.print(h);
  Serial.print(" %\t");
  Serial.print("Temperatura: ");
  Serial.print(t);
  Serial.println(" C");

}
```

En monitor de puerto serie del entorno de desarrollo vemos los datos de temperatura y humedad adquiridos del sensor digital.

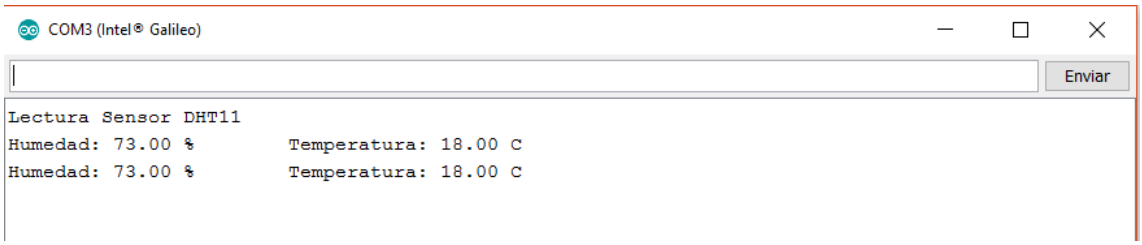

Ilustración 54: Monitorización de puerto serie de la ***¡Error! No se encuentra el origen de la referencia.****. Fuente Propia.*

7.1.3. Comunicación CAN BUS

Para este apartado solo vamos a conectar el CAN Bus Shield a la placa y comprobaremos su funcionamiento utilizando el monitor de línea serie.

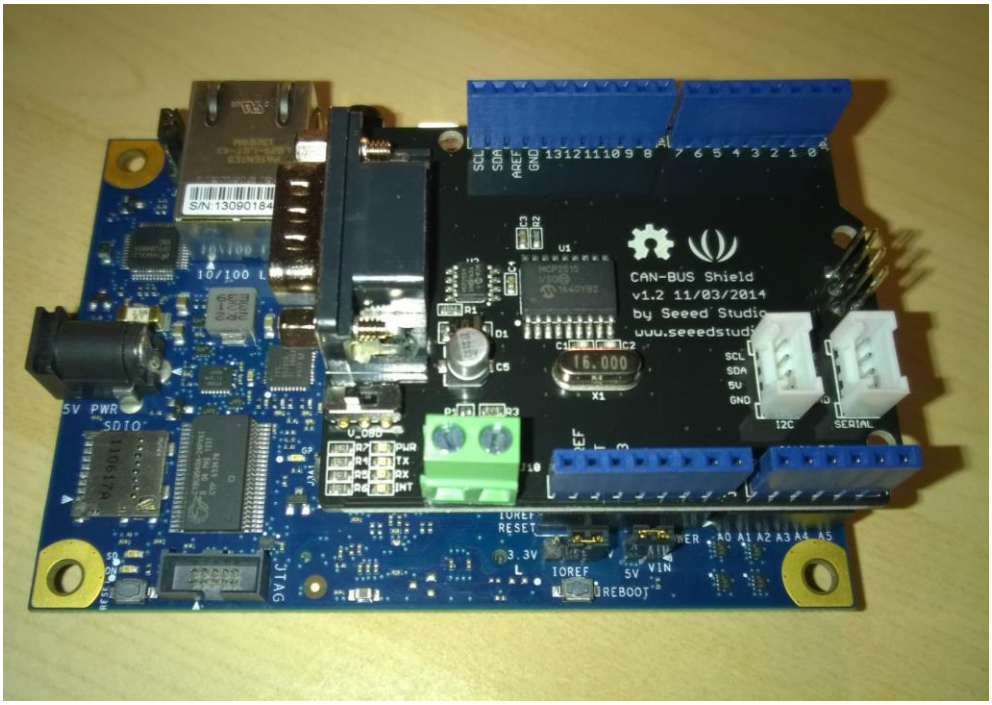

Ilustración 55: Montaje Shield Can Bus. Fuente Propia

Para poder realizar la programación de la comunicación CAN Bus por parte del nodo esclavo necesitamos utilizar la librería que acompaña al Shield CAN Bus [69]. También existen librerías compatibles con cualquier Shield que utilice el integrado MCP2515 o el integrado MCP25625, en caso de utilizar otros dispositivos [70].

Se ha tomado esta decisión para que este proyecto pueda ser reproducido con cualquier otro shield disponible en ese momento en el mercado.

Primero incluimos la librería comentada anteriormente, así como la librería que nos permite utilizar el puerto SPI. Creamos una instancia de tipo MCP_CAN conectado al puerto 9, aunque también puede conectarse al puerto 10.

Para esta prueba hemos generado una trama de datos de 8 bytes que consiste en un array de números del 1 al 8.

```
#include <mcp_can.h>
#include <mcp_can_dfs.h>

#include <SPI.h>

const int SPI_CS_PIN=9;

MCP_CAN CAN_ESCLAVO(SPI_CS_PIN);

unsigned char datos[8] = {1,2,3,4,5,6,7,8};;

void setup() {
  Serial.begin(115200);

  while (CAN_ESCLAVO.begin(CAN_500KBPS) != CAN_OK) {
    Serial.println("Error de inicialización");
    delay(100);
  }
  Serial.println("Comunicación Inicializada correctamente");
}

void loop() {
  byte estadoEnvio = CAN_ESCLAVO.sendMsgBuf(0x10, 0, 8, datos);

  if (estadoEnvio == CAN_OK) {
    Serial.println("Mensaje Enviado Correctamente");
  } else {
    Serial.println("Error Enviando Mensaje");
  }
  delay(100);
}
```

Posteriormente iniciamos el puerto serie a 115200 baudios y la comunicación CAN a 500 KBPS.

Mostramos un mensaje por pantalla para indicar si la inicialización se ha realizado correctamente o no. En caso negativo repetimos la inicialización tras 100ms.

Fijamos el modo de trabajo a normal para permitir que los mensajes se envíen.

Ya dentro de la función bucle, enviamos la trama fijada cada 100 ms y mostramos por el puerto serie si el envío ha sido correcto o no.

Ilustración 56: Monitorización de puerto serie de la comunicación CAN Bus del nodo esclavo. Fuente Propia

7.1.4. Comunicación Ethernet

En este punto de la memoria se va a proceder a implementar las dos partes iniciales de la comunicación Ethernet [71]. En capítulos posteriores, cuando se implemente el diseño completo, ya añadiremos la adquisición de datos de los sensores y el envío de éstos utilizando el protocolo Ethernet.

Una de las partes es asignar una dirección IP ya sea dinámica o estática a la placa Intel Galileo y una vez configurada su IP, procederemos a la creación de un servidor web para que el PC pueda conectarse a la placa Intel y obtener los valores que está midiendo [72].

Asignación de IP

Para este apartado, como se ha explicado detenidamente en el capítulo 6.4, conectando la placa Intel Galileo “esclavo” a un router, y, mediante el monitor de puerto serie vamos a comprobar simplemente si podemos inicializar una comunicación Ethernet [73] .

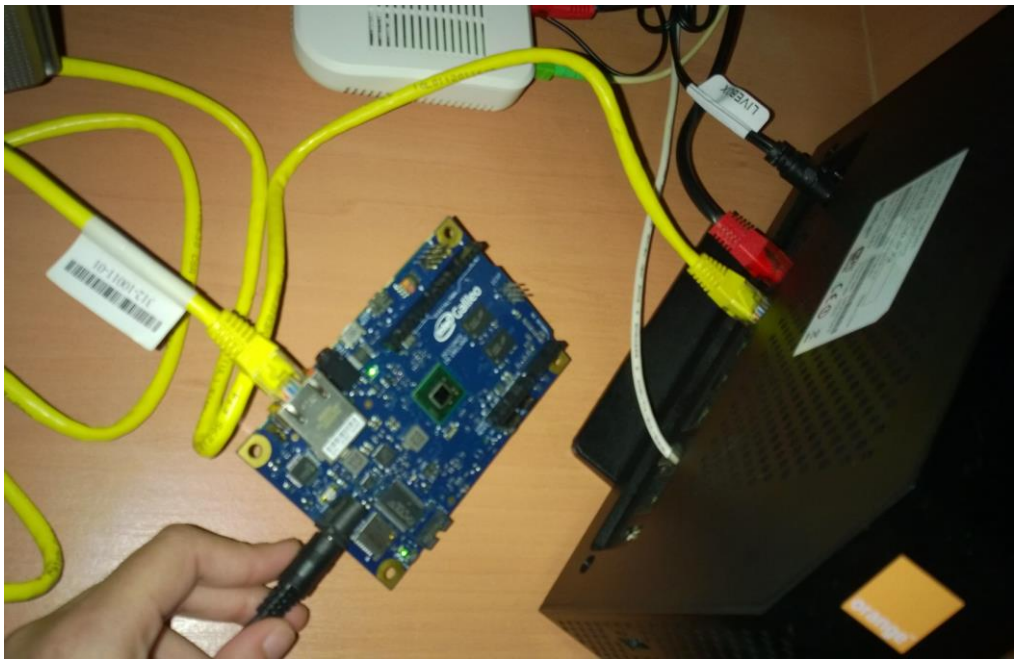

Ilustración 57: Conexión de la placa Intel Galileo al router. Fuente propia

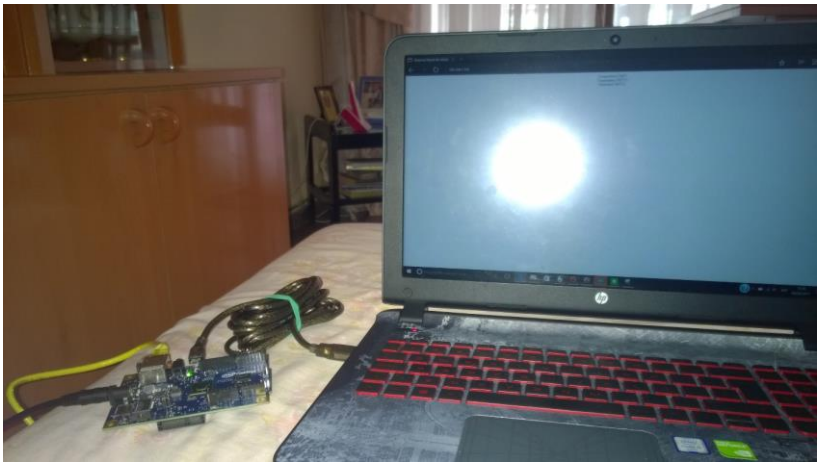

Ilustración 58: Conexión de la placa Intel Galileo al PC. Fuente Propia

Primero de todo necesitamos obtener la Mac de la placa Intel Galileo. Esta información se encuentra en una pegatina en la capa inferior de la placa.

Ilustración 59: Localización de la dirección MAC de una Intel Galileo. Fuente Propia

En caso de que el router que vayamos a utilizar no presente el protocolo DHCP[14] [74], tenemos que indicar en el sketch una IP estática dentro de nuestra red local para que sea asignada a la placa.

Para ver la red local que estamos utilizando, desde un ordenador conectado al mismo router abrimos el terminal (símbolo de sistema si estamos en Windows) y ejecutamos el siguiente comando:

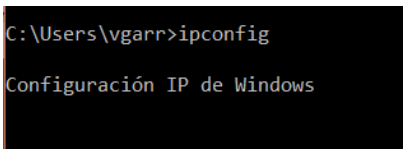

Ilustración 60: Configuración IP de Windows. Fuente Propia

En caso de estar utilizando Linux el comando es ifconfig.

Al ejecutar el comando nos aparece la configuración de todos los adaptadores de red y máquinas virtuales (en caso de tener alguna instalada en el pc). Si estamos conectados vía Ethernet tenemos que mirar los datos que aparecen en el apartado Adaptador de Ethernet, en caso de estar conectados vía Wifi tenemos que mirar el apartado Adaptador de LAN. Independientemente el contenido es el mismo:

[14] Protocolo de configuración de host dinámico.

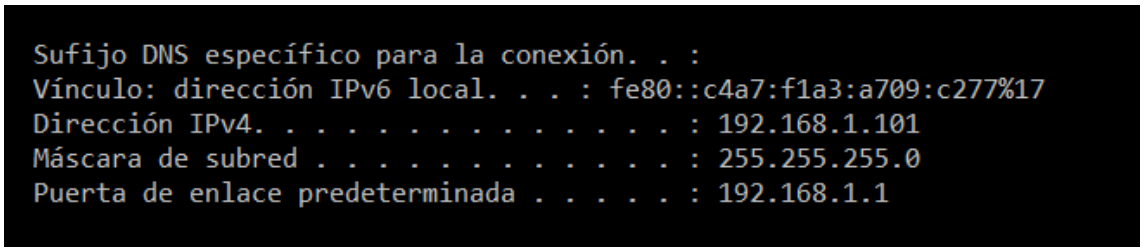

Ilustración 61: Dirección IP del adaptador de Ethernet. Fuente propia

Como podemos observar, en este caso nuestra red local es la 192.168.1.X, con lo cual dejaremos como IP estática, en caso de que nos falle el protocolo DHCP la dirección 192.168.1.177.

Para ello es necesario ejecutar el siguiente sketch:

```
#include <Dhcp.h>
#include <Dns.h>
#include <Ethernet.h>
#include <EthernetClient.h>
#include <EthernetServer.h>
#include <EthernetUdp.h>

#include <SPI.h>
 byte mac[]={0x00,0x13,0x20,0xFF,0x14,0x0C};
 IPAddress ip(192,168,1,103);

void setup() {
  Serial.begin(115200);
  delay(5000);
  Serial.println("Configurando la red Ethernet usando DHCP");
  if(Ethernet.begin(mac)==0){

    Serial.println("Error al configurar la red Ethernet usando DHCP");
    Serial.println("Configurando Ethernet usando una IP estática");
    Ethernet.begin(mac,ip);
  }else{
    Serial.println("Genial");
  }
  delay(1000);
  Serial.print("Mi direccion IP es ");
  Serial.println(Ethernet.localIP());
  //system("ifup eth0");

}

void loop() {
  delay(1000);
  Serial.print("Mi direccion IP es ");
  Serial.println(Ethernet.localIP());
  //system("ifup eth0");

}
```

El código anterior primero trata de inicializar la placa Intel utilizando los DNS, en caso de que se produzco un error, se configura la IP manualmente, en caso contrario se muestra por el puerto serie la dirección IP que se le ha asignado dinámicamente.

Utilizando el monitor de puerto serie vemos como se ha configurado y su nueva IP:

```
Configurando la red Ethernet usando DHCP
Genial
Mi direccion IP es 192.168.  1.103
Mi direccion IP es 192.168.  1.103
Mi direccion IP es 192.168.  1.103
Mi direccion IP es 192.168.  1.103
Mi direccion IP es 192.168.  1.103
```

Ilustración 62: Monitorización del puerto serie del sketch de la ***¡Error! No se encuentra el origen de la referencia.****. Fuente propia.*

Creación de un servidor Ethernet

El siguiente código actúa como servidor, es decir, devuelve una página web cuando un cliente, en este caso el PC, se conecta a él.

Para esta prueba, vamos a mostrar una página web con simplemente texto. Para ello necesitamos enviar por el cliente Ethernet cadenas de texto html.

```
#include <Dhcp.h>
#include <Dns.h>
#include <Ethernet.h>
#include <EthernetClient.h>
#include <EthernetServer.h>
#include <EthernetUdp.h>

#include <SPI.h>

byte mac[] = {0x00, 0x13, 0x20, 0xFF, 0x14, 0x0C};
IPAddress ip(192, 168, 1, 177);
EthernetServer servidor(80);

void setup() {
  Serial.begin(115200);
  if (Ethernet.begin(mac) == 0) {
  Ethernet.begin(mac, ip);
  }
  delay(1000);
  servidor.begin();
  Serial.print("Mi dirección IP es ");
  Serial.println(Ethernet.localIP());
```

```
  delay(100);
  dht.begin();
}

void loop() {
  EthernetClient cliente = servidor.available();
  bool isOk;
  float humedad, temperatura;
  if (cliente) {
    // Serial.println("Nuevo cliente");
    bool lineaBlanca = true;
    //String cadena = "";
    while (cliente.connected()) {
      if (cliente.available()) {
        char c = cliente.read();
        if (c == '\n' && lineaBlanca) {
          cliente.println("HTTP/1.1 200 OK"); //Encabezadp de HTTP
estándar
          cliente.println("Content-Type:text/html");
          cliente.println("Connection:close"); //La sesion se cerrarña
después de enviar la respueta
          cliente.println("Refresh:20"); //Refrescar automáticamente
la página cada 20 segundos
          cliente.println();
          cliente.println("<!DOCTYPE HTML>"); //Tipo de documento
          cliente.println("<html>"); //Inicio de documento
          cliente.println("<head>\n<title>Sistema Naval de adquisicion
de datos </title>\n</head>"); //Titulo de la pagina
          cliente.println("<div style='text-align:center'>");
          cliente.println("<body>");//Inicio del cuerpo
          cliente.println("<H1>Temperatura LM35</H1>");
          cliente.println("<br/>"); //Salto de línea
          cliente.println("<H1>Humedad DHT11</H1>");
          cliente.println("<br/>"); //Salto de línea
          cliente.println("<H1>Temperatura DHT11</H1>");
          cliente.println("<br/>"); //Salto de línea
          cliente.println("</body>"); //Fin del cuerpo
          cliente.println("</html>"); //Fin del documento
          break;
        }
      }
    }
    delay(1);
    cliente.stop();
    Serial.println("Cliente desconectado");
  }

}
```

Utilizando el monitor de puerto serie, vamos a obtener la IP que a la que ha sido asignada la placa y vamos a debuggear el sketch viendo si recibimos los mensajes de conexión y desconexión del cliente que hemos añadido.

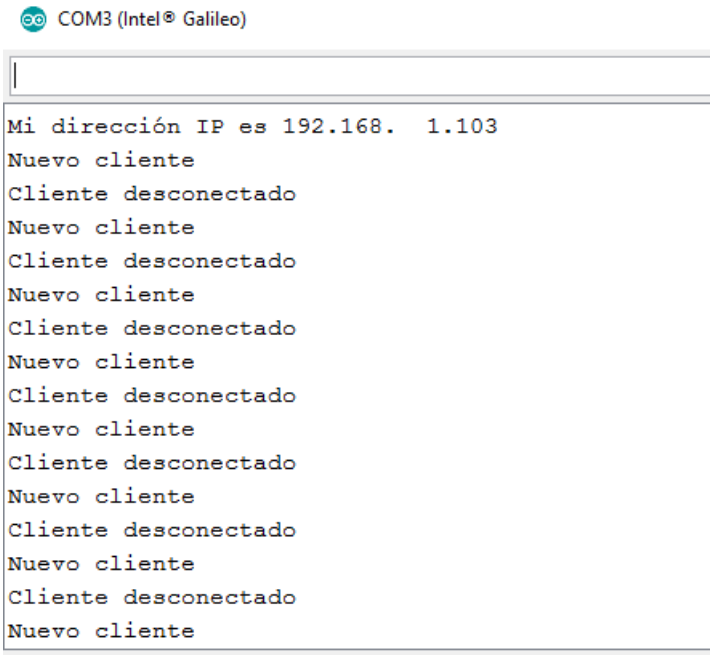

Ilustración 63: Monitorización del puerto del servidor Web. Fuente Propia

Para visualizar la página simplemente accedemos a la dirección IP que vemos en la Ilustración 63 desde un navegador web, estando el PC conectado en la misma red local que la placa Intel Galileo.

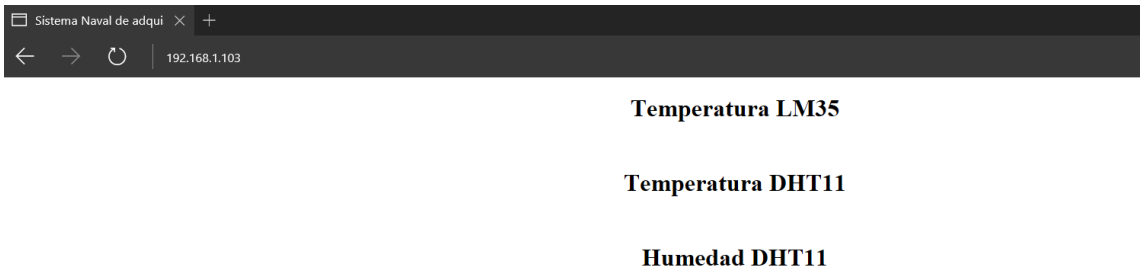

Ilustración 64: Página web creada con la placa Intel Galileo. Fuente propia

7.2. Nodo Maestro

El nodo maestro es aquella placa Intel Galileo que se va a encargar de leer los datos enviados por CAN desde el nodo esclavo.

7.2.1. Comunicación CAN Bus

Conectaremos la placa Intel Galileo al shield tal cual lo hemos realizado en el apartado 7.1.3. Además, interconectaremos ambas placas Intel utilizando dos cables respetando siempre la paridad CAN_H y CAN_L, es decir, conectaremos el pin CAN_H del nodo esclavo con el CAN_H del nodo maestro y lo mismo respectivamente para el CAN_L.

```
#include <mcp_can.h>
#include <mcp_can_dfs.h>

#include <SPI.h>

const int SPI_CS_PIN=9;
MCP_CAN CAN(SPI_CS_PIN);

void setup() {
  Serial.begin(115200);
  while (CAN_OK!=CAN.begin(CAN_500KBPS))
  {
    Serial.println("Error de inicializacion");
    delay(100);
  }
  Serial.println("Comunicacion inicializada correctamente");
}
void loop() {
  unsigned char len=0;
  unsigned char buf[8];
  if (CAN.checkReceive() == CAN_MSGAVAIL) {
    CAN.readMsgBuf(&len, buf);
    unsigned int canId=CAN.getCanId();
    Serial.print("ID: ");
    Serial.println(canId,HEX);
    for (int i = 0; i < len; i++) {
      Serial.print(buf[i],HEX);
      Serial.print("\t");
    }
    Serial.println();
  }
}
```

Comenzamos inicializando de igual manera el shield para en el bucle comprobar si tenemos algún mensaje disponible en el buffer. En caso afirmativo lo leemos y lo mostramos por pantalla.

El montaje es exactamente el mismo que el mostrado en la Ilustración 55 pero utilizando los elementos asignados para formar parte del nodo maestro.

Si ejecutamos el sketch, sin conectar el nodo esclavo simplemente nos mostrará el mensaje de inicialización correcta del módulo.

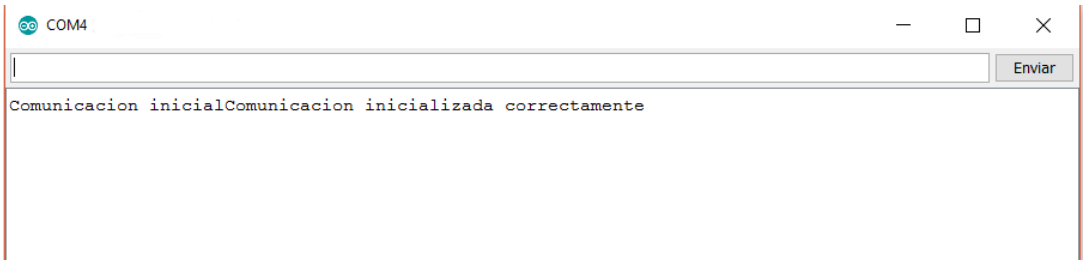

Ilustración 65: Monitorización de puerto serie del Nodo Master. Fuente Propia

Queda pendiente para el próximo apartado en función de la id del mensaje detectar que datos estamos leyendo.

7.3. Implementación Diseño Completo

7.3.1. Sistema Completo Ethernet

Montaje

Una vez comprobado el correcto funcionamiento del código por separado de la creación del servidor Ethernet y de la lectura de los sensores, pasamos a unirlo todo para tener implementado el diseño del capítulo 6.4.

Para realizar el montaje simplemente conectamos la protoboard con el cableado de los sensores LM35 y DHT11, según se muestra en la Ilustración 51 y la Ilustración 53 a la Intel Galileo y ésta al router.

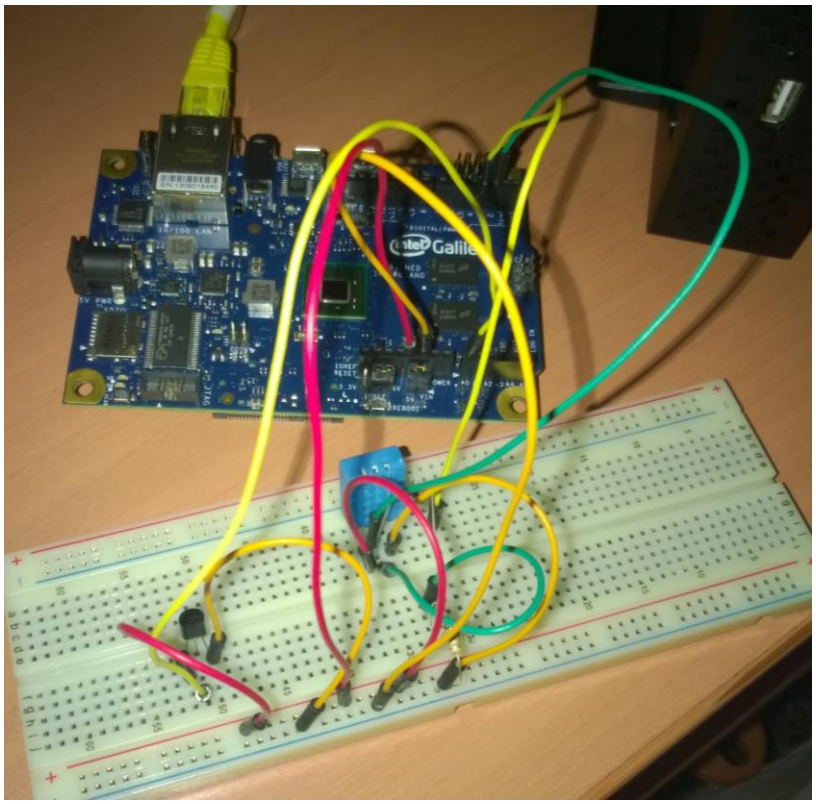

Ilustración 66: Montaje implementación del diseño completo utilizando Ethernet. Fuente propia

Código

```
#include <DHT.h>

#include <Dhcp.h>
#include <Dns.h>
#include <Ethernet.h>
#include <EthernetClient.h>
#include <EthernetServer.h>
#include <EthernetUdp.h>

#include <SPI.h>

#define DHTIN 2
#define DHTOUT 3
#define DHTTYPE DHT11

byte mac[] = {0x00, 0x13, 0x20, 0xFF, 0x14, 0x0C};
IPAddress ip(192, 168, 1, 177);
EthernetServer servidor(80);

DHT dht(DHTIN, DHTOUT, DHTTYPE);

void setup() {
  Serial.begin(115200);
  if (Ethernet.begin(mac) == 0) {
  Ethernet.begin(mac, ip);
  }
  delay(1000);
  servidor.begin();
  Serial.print("Mi dirección IP es ");
  Serial.println(Ethernet.localIP());
  delay(100);
  dht.begin();
}

void loop() {
  EthernetClient cliente = servidor.available();
  bool isOk;
  float humedad, temperatura;
  if (cliente) {
    Serial.println("Nuevo cliente");
    bool lineaBlanca = true;
    while (cliente.connected()) {
      if (cliente.available()) {
        char c = cliente.read();
        if (c == '\n' && lineaBlanca) {
          cliente.println("HTTP/1.1 200 OK"); //Encabezadp de HTTP
estándar
          cliente.println("Content-Type:text/html");
          cliente.println("Connection:close"); //La sesion se cerrarña
después de enviar la respueta
          cliente.println("Refresh:20"); //Refrescar automáticamente
la página cada 20 segundos
          cliente.println();
          cliente.println("<!DOCTYPE HTML>"); //Tipo de documento
          cliente.println("<html>"); //Inicio de documento
          cliente.println("<head>\n<title>Sistema Naval de adquisicion
de datos </title>\n</head>"); //Titulo de la pagina
          cliente.println("<div style='text-align:center'>");
```

```
            cliente.println("<body>");//Inicio del cuerpo
            cliente.println("<pre>");
            cliente.println("<H1>Temperatura LM35</H1>");
            //Insertar dato de la temperatura
            float temperatura35 = getSensor35();
            cliente.println("<H2>");
            cliente.println(temperatura35);
            cliente.println("</H2>");
            cliente.println("<br/>"); //Salto de línea
            getSensorDHT(&humedad, &temperatura, &isOk);
            cliente.println("<H1>Temperatura DHT11</H1>");
            //Insertar dato de la temperatura del dht11
            cliente.println("<H2>");
            if  (isOk == true) {
              cliente.println(temperatura);
            } else {
              cliente.println("Error de lectura");
            }
            cliente.println("</H2>");
            cliente.println("<br/>"); //Salto de línea
            cliente.println("<H1>Humedad DHT11</H1>");
            //Insertar dato de la humedad

            cliente.println("<H2>");
            if  (isOk == true) {
              cliente.println(humedad);
            } else {
              cliente.println("Error de lectura");
            }
            cliente.println("</pre>"); //Salto de línea
            cliente.println("</H2>");
            cliente.println("<br/>"); //Salto de línea
            cliente.println("</body>"); //Fin del cuerpo
            cliente.println("</html>"); //Fin del documento
            break;
          }
          if (c == '\n') {
            lineaBlanca = true;
          } else if (c != '\r') {
            lineaBlanca = false;
          }
        }
      }
      delay(1);
      cliente.stop();
      //Serial.println("Cliente desconectado");
    }

}

float getSensor35() {
  const int LM35Pin = A0;
  int lecturaLM35 = analogRead(LM35Pin);
  float milivoltios = (lecturaLM35 / 1023.0) * 5000;
  float grados = milivoltios / 10;
  //Serial.print(grados);
  //Serial.println("°C");
  return grados;
}

void getSensorDHT(float *h, float *t, bool *isOk) {
```

```
  *isOk = true;
  *h = dht.readHumidity();
  *t = dht.readTemperature();
  if (isnan(*h) || isnan(*t)) {
    *isOk = false;
  }
}
```

Hemos dividido el código en 4 funciones [75]:

- Setup

En esta función, tras la declaración de las variables globales y del objeto dht, asignamos a la placa una dirección IP (en este caso se ha dejado la asignación estática, pero si se desea una asignación estática, solo hay que comentar el código. Posteriormente se inicializa el servidor, así como el objeto dht.

- Loop

Esta es la función que se ejecutará en bucle. Es el mismo código que el utilizado para crear el servidor, sin embargo, presenta una serie de cambios y mejoras.

El cambio de refresco se ha aumentado a 20 segundos puesto que con 5 segundos se daba el caso de que no se refrescaba correctamente los valores obtenidos del sensor DHT11. Esto es debido al procesamiento tan lento que presenta esta familia de sensores.

También se han comentado las líneas de los mensajes serie que utilizamos en el capítulo Creación de un servidor Ethernet para depurar y comprobar el funcionamiento del código.

- getSensor35

Hemos creado esta función para obtener el valor de la temperatura que se obtenía a partir del sensor LM35.

Esta función devuelve un float. Es decir, cada vez que llamemos a la función nos devolverá la temperatura en grados.

- GetSensorDHT

Esta función tiene sus particularidades. Tenemos que devolver dos parámetros (la humedad y la temperatura) sin embargo C++ no permite los return de múltiples parámetros [76]. La forma correcta de hacerlo es utilizar punteros y modificar el contenido de ellos dentro de la función sin devolverlos explícitamente.

Debido a que a veces no se leen correctamente los valores del sensor, se le pasa un booleano [77] que tomará el valor de true en caso de tener una lectura válida.

En función del valor de este booleano en la función loop, el cliente web muestra el valor o un mensaje de error.

Otra manera de realizar esta función (y también sería válida) es que solo reciba como parámetro humedad y temperatura y que isOk sea el return de la función.

Prueba Final

En nuestro caso, para facilitar esta prueba, hemos dejado la asignación de IP estática a la dirección 192.168.1.177.

Poniendo esta dirección en un navegador web, creamos un nuevo cliente que se actualiza solo cada 20 segundos y que se muestra tal cual sigue:

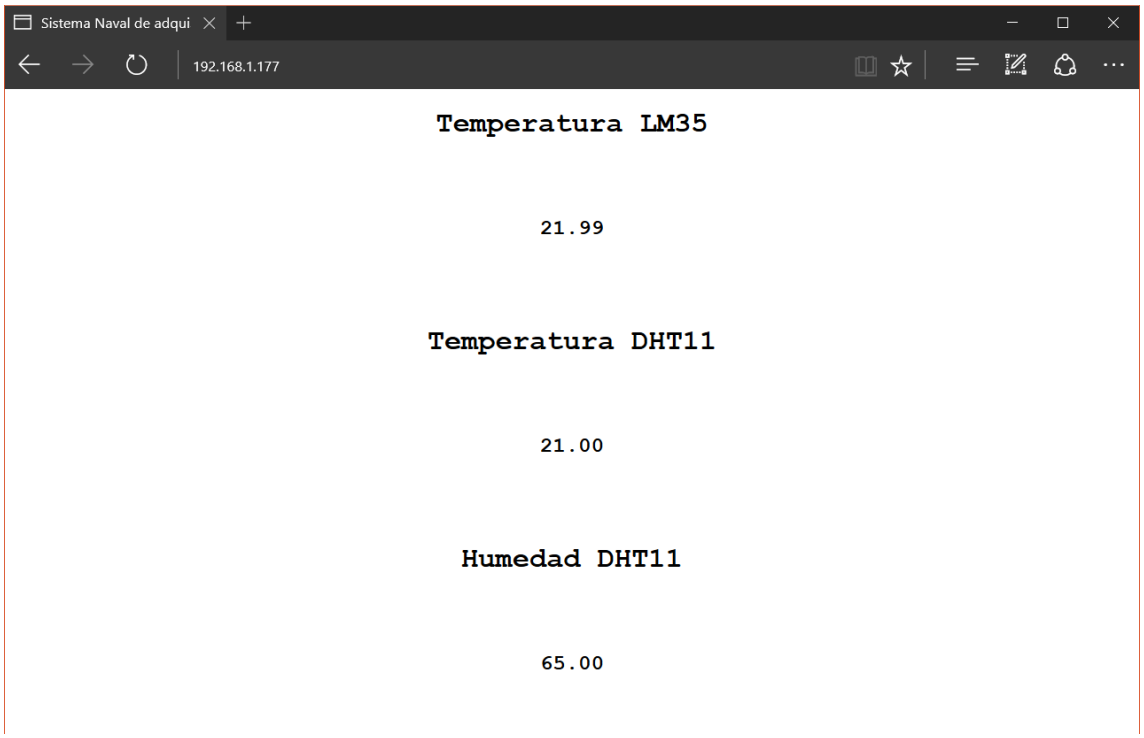

Ilustración 67: Pagina web diseño completo Ethernet. Fuente Propia

7.3.2. Sistema Completo CAN Bus

Montaje

Tras verificar la implementación del código del envío y recepción de mensajes CAN (capítulos 7.1.3 y 7.2.1) así como de la lectura de los sensores, en este capítulo realizamos la fusión del código para implementar el diseño estipulado en el capítulo 6.3.

Para realizar el montaje primero conectamos la protoboard con el cableado de los sensores LM35 y DHT11, según se muestra en la Ilustración 51 y la Ilustración 53. Tras esto hay que situar sobre la Intel Galileo el shield que nos permite tener comunicación CAN Bus.

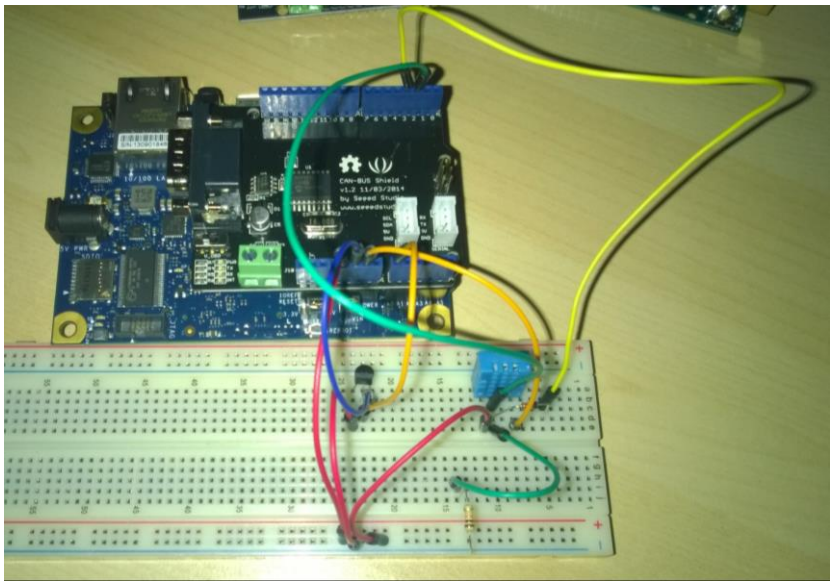

Ilustración 68: Montaje nodo esclavo para implementación de diseño completo. Fuente Propia

Faltaría interconectar ambos nodos. Existen dos formas de hacerlo, a través de un conector hembra-hembra DB9 o con mediante cables utilizando los conectores.

Ilustración 69: Conexión entre nodo esclavo y nodo master. Fuente propia.

Código nodo esclavo

```
#include <DHT.h>

#include <SPI.h>

#include <mcp_can.h>
#include <mcp_can_dfs.h>
#define DHTIN 2
#define DHTOUT 3
#define DHTTYPE DHT11

DHT dht(DHTIN, DHTOUT, DHTTYPE);
const int SPI_CS_PIN = 9;
MCP_CAN CAN_ESCLAVO(SPI_CS_PIN);
unsigned char datos[8];

void setup() {
  Serial.begin(115200);

  while (CAN_ESCLAVO.begin(CAN_500KBPS) != CAN_OK) {
    Serial.println("Error de inicialización");
    delay(500);
  }
  Serial.println("Comunicación Inicializada correctamente");
  dht.begin();
}

void loop() {
  float humedad, temperatura;
  float temperatura35 = getSensor35();
  bool isOk;
  isOk = getSensorDHT(&humedad, &temperatura);
  memcpy(&datos, &temperatura35, sizeof(datos));
  //Enviamos la temperatura del sensor LM35
  byte estadoEnvio = CAN_ESCLAVO.sendMsgBuf(1, 0, 8, datos);
  if (estadoEnvio == CAN_OK) {
    Serial.println("Mensaje Enviado Correctamente");
  } else {
    Serial.println("Error Enviando Mensaje");
  }
  if (isOk == true) {
    //Enviamos la temperatura de la humedad del sensor DHT11
    memcpy(&datos, &humedad, sizeof(datos));
    estadoEnvio = CAN_ESCLAVO.sendMsgBuf(2, 0, 8, datos);
    if (estadoEnvio == CAN_OK) {
      Serial.println("Mensaje Enviado Correctamente");
    } else {
      Serial.println("Error Enviando Mensaje");
    }
    //Enviamos la temperatura de la temperatura del sensor DHT11
    memcpy(&datos, &temperatura, sizeof(datos));
    estadoEnvio = CAN_ESCLAVO.sendMsgBuf(3, 0, 8, datos);
    if (estadoEnvio == CAN_OK) {
      Serial.println("Mensaje Enviado Correctamente");
    } else {
      Serial.println("Error Enviando Mensaje");
    }
  }
  else {
```

```
    memcpy(&datos, NULL, sizeof(datos));
    for (int i = 0; i < 2; i++) {
      CAN_ESCLAVO.sendMsgBuf(i+2, 0, 8, datos);
      delay(20);
    }

  }
  delay(100);

}

float getSensor35() {
  const int LM35Pin = A0;
  int lecturaLM35 = analogRead(LM35Pin);
  float milivoltios = (lecturaLM35 / 1023.0) * 5000;
  float grados = milivoltios / 10;
  return grados;
}

bool getSensorDHT(float *h, float *t) {
  bool isOk = true;
  *h = dht.readHumidity();
  *t = dht.readTemperature();
  if (isnan(*h) || isnan(*t)) {
    isOk = false;
  }
  return isOk;
}
```

Al igual que se ha realizado en el caso de la implementación del sistema completo con Ethernet, aquí también se ha separado el código en 4 funciones distintas más la declaración de componentes.

- Declaración de componentes

Como hemos realizado en otros sketches, incluimos las librerías de SPI, CAN Bus y la librería del sensor DHT y creamos la instancia de CAN en el pin 9 y la instancia del sensor DHT.

- Setup

Esta función es idéntica a la utilizada en el capítulo 7.1.3.

Inicializamos la comunicación serie (para comprobar el funcionamiento) y la comunicación CAN mostrando por el monitor de puerto serie el resultado de esta inicialización.

- Loop

Llamando a las funciones getSensor35() y getSensorDHT leemos los valores de los sensores (y se procesan a su unidad correspondiente). Estas dos funciones se han explicado detalladamente en el capítulo 7.3.1.

El protocolo CAN que estamos utilizando es el básico, el cual envía tramas de 11 bytes, con lo cual, si descartamos los 3 bytes de información de identificador, tipo de trama y longitud de trama, nos queda que los datos que tenemos que enviar son 8 bytes de tipo char.

Como los datos que tenemos que enviar en nuestro caso son de tipo float, realizamos una copia de este valor en la dirección de memoria a la que apunta el buffer de envío de datos.

En nuestro caso, al tener una estructura fija de 3 señales se ha procedido a dar un Id distinto a cada una para identificarla en el nodo master. En caso de trabajar con más señales o más dispositivos que implicaría tener que rechazar este método, la forma correcta sería en lugar de enviar el float del sensor, enviar una estructura de 8 bytes formada por el float y un tagname asociado a cada canal de lectura.

En el capítulo 7.1.2 detectamos que a veces perdíamos la señal del sensor recibiendo el valor de NaN. Este caso también se ha contemplado aquí y si el booleano isOk es falso, es decir, si alguna lectura es NaN enviamos dos (uno por cada sensor) NULL al master para que lo procese igualmente.

Código nodo master

```
#include <mcp_can.h>
#include <mcp_can_dfs.h>

#include <SPI.h>

const int SPI_CS_PIN = 9;
MCP_CAN CAN(SPI_CS_PIN);

void setup() {
  Serial.begin(115200);
  while (CAN_OK != CAN.begin(CAN_500KBPS))
  {
    Serial.println("Error de inicializacion");
    delay(100);
  }
  Serial.println("Comunicacion inicializada correctamente");
```

```
}
void loop() {
  unsigned char len = 8;
  unsigned char buf[8];
  if (CAN.checkReceive() == CAN_MSGAVAIL) {
    CAN.readMsgBuf(&len, buf);
    unsigned int canId = CAN.getCanId();
    Serial.print("ID: ");
    Serial.println(canId);
    switch (canId) {
      case 1:
        Serial.print("Temperatura LM35: ");
        break;
      case 2:
        Serial.print("Humedad DHT11: ");
        break;
      case 3:
        Serial.print("Temperatura DHT11: ");
        break;
      default:
        Serial.print("Error ");
        break;
    }
    if (buf == NULL) {
      Serial.println ("Error");
    } else {
      for (int i = 4; i < len; i++) {       //Los float solo ocupan 4
bytes
        if (i != 5 & i!6) {
          Serial.print(buf[i]);
          Serial.print("\t");
        } else if (i == 0) {
          i++;
        }

      }
      if (canId == 1 || canId == 3) {
        Serial.println("ยบC");
      } else if (canId == 2) {
        Serial.println("%");
      }
    }
  }
  delay(100);
}
```

En este caso solo tenemos las dos funciones básicas setup y loop.

- Setup

Se inicializa la comunicación CAN Bus y el puerto serie para la realización de las pruebas pertinentes.

- Loop

En bucle estamos comprobando si hemos recibido algún mensaje. En caso de tener un mensaje disponible para su procesado, almacenamos su contenido en un char de 8 bytes, es decir, en un buffer.

Obtenemos también la Id del mensaje y en función del valor que tome identificamos el tipo de mensaje que hemos recibido.

A la hora de mostrar por pantalla el valor del buffer, eliminamos los 4 primeros bytes ya que estos son todo cero (recordemos que un float solo ocupa 4 bytes de memoria) y recorremos el buffer hasta el final.

Prueba final

Conectando la Intel Galileo designada como nodo maestro al ordenador, y utilizando el monitor de puerto serie integrado en el IDE de Arduino hemos obtenido el siguiente resultado:

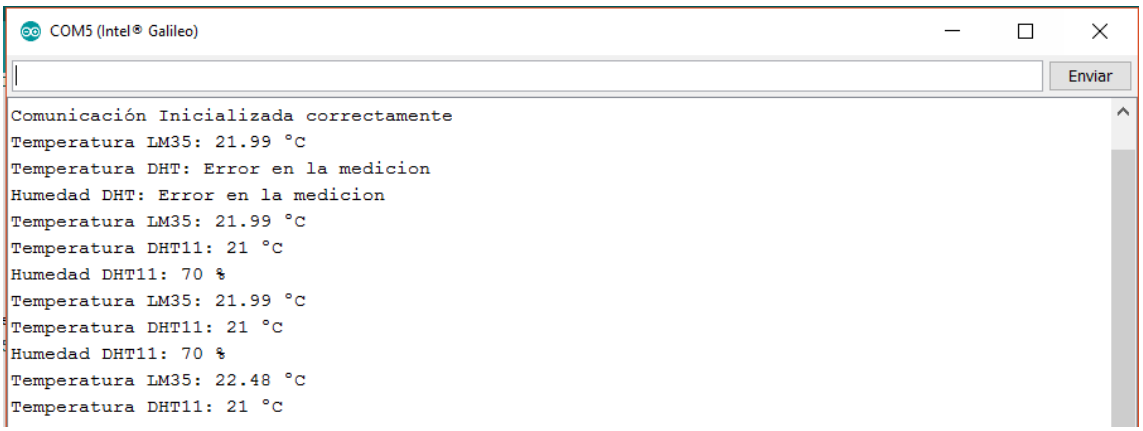

Ilustración 70: Prueba final implementación completa CAN Bus. Fuente Propia

8. Presupuesto

Primero de todo procederemos a calcular el presupuesto por partidas concluyendo con un resumen de ellas que proporcionará el coste final de este proyecto.

8.1. Presupuesto por Partidas

	CANTIDAD	PRECIO UD.	TOTAL
Intel Galileo	2	82.50€	165€
CAN Bus Shield	2	34.46€	68.92€
USB A – micro USB B	2	2.25€	4.5€
Diodo 1N4148	1	0.04€	0.04€
Resistencias 10K	1	0.03€	0.03€
Sensor LM35	1	1.50€	1.50€
Sensor DHT11	1	2.80€	2.80€
Router Modem	1	22€	22€
Cables Macho-Macho	1	2.04€	2.04€
Protoboard	1	2.80€	2.80€
TOTAL			269.63€

Tabla 11: Presupuesto partida material. Fuente Propia

Puesto	Cantidad	Horas	Salario	Importe
Ingeniero de Telecomunicación	1	360	8€	2880€

Tabla 12: Presupuesto partida mano de obra. Fuente Propia

8.2. Presupuesto total

Partida	Precio
Partida material	269.63€
Partida mano de obra	2880€
TOTAL	3149.63€

Tabla 13: Presupuesto total. Fuente Propia.

9. Conclusiones

A lo largo de este Trabajo Final de Master se ha desarrollado el diseño de un sistema de adquisición de datos para el sistema naval cumpliendo con éxito todos los objetivos propuestos.

El estudio comenzó desde una comparativa de SoC disponibles en el mercado hasta el diseño de un modelo que pueda trabajar con el protocolo de comunicación CAN Bus y una alternativa basada en Ethernet pasando por el estudio teórico del funcionamiento básico de estos protocolos.

El interés de este tipo de sistema de adquisición no solo radica en la necesidad de modernizar los sistemas de la industria naval si no en la tendencia que predice un auge en un futuro donde se sustituya el cableado CAN por un cable Ethernet.

Tanto la metodología como la planificación utilizada en este trabajo final de master se han seguido sin problemas y han ocasionado que el desarrollo fuese tal cual lo planeado.

Como futura línea de trabajo se propone fusionar ambos diseños en un uno, es decir, que se comuniquen los nodos por protocolo CAN y Ethernet al mismo tiempo. Otra línea posible podría ser el diseño de una carcasa que proporcione aislamiento al diseño frente a las vibraciones y la humedad.

10. Glosario

A

ADC
Conversor Analógico Digital 17

Arduino
Es una compañía de hardware libre y una comunidad tecnológica que diseña y manufactura placas computadora de desarrollo de hardware y software, compuesta respectivamente por circuitos impresos que integran un microcontrolador y un entorno de desarrollo (IDE), en donde se programa cada placa. 23

B

bit
Es un dígito del sistema de numeración binario 49

byte
Es la unidad de información de base utilizada en computación y en telecomunicaciones, y que resulta equivalente a un conjunto ordenado de 8 bits 63

C

CAN Bus
Can-Bus es un protocolo de comunicación en serie desarrollado por Bosch para el intercambio de información entre unidades de control electrónicas del automóvil. VII

codificación Manchester
Es un método de codificación eléctrica de una señal binaria en el que en cada tiempo de bit hay una transición entre dos niveles de señal 68

CRC
Comprobación de Redundancia Cíclica 63, 64, 70

D

DAC
Conversor Digital a Analógico 17

datasheet
Es un documento que resume el funcionamiento y otras características de un componente 37

dirección IP
Es un número que identifica, de manera lógica y jerárquica, a una Interfaz en red (elemento de comunicación/conexión) de un dispositivo. 106

E

Eclipse
Es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores 36

Ethernet
Es un estándar de redes de área local para computadores con acceso al medio por detección de la onda portadora y con detección de colisiones. 8

H

Hardware
Partes físicas tangibles de un sistema informático 26, 27

I

M

N

O

P

R

S

T

11. Bibliografía

[1] «¿Qué es Adquisición de Datos? - National Instruments». [En línea]. Disponible en: http://www.ni.com/data-acquisition/what-is/esa/. [Accedido: 17-mar-2017].

[2] «Teorema de muestreo de Nyquist-Shannon», *Wikipedia, la enciclopedia libre*. 11-feb-2017.

[3] W. Kester, «Analog-Digital Conversion», en *Data Conversion Handbook*, Analog Devices Training Seminars.

[4] K. Cecen, *Sinan's water supply system in Istanbul*. ISKI, Istanbul Su ve Kanalizasyon Idaresi Genel Mudurlugu, 1992.

[5] F. M. Smits, Ed., *A History of Engineering and Science in the Bell System: Electronics Technology*, 1st edition. Indianapolis: AT&T Laboratories, 1985.

[6] H. F. Mayer, «Principles of Pulse Code Modulation», en *Advances in Electronics and Electron Physics*, vol. 3, L. Marton, Ed. Academic Press, 1951, pp. 221-260.

[7] P. Rainey, «Facsimile telegraph system».

[8] R. A. Harley, «Electric signaling system», feb-1942.

[9] J. S. Kilby, «Invention of the integrated circuit», *IEEE Trans. Electron Devices*, vol. 23, n.º 7, pp. 648-654, jul. 1976.

[10] R. Pallás-Areny, *Sensores y acondicionadores de señal*. Barcelona; México, D.F.: Marcombo ; Alfaomega, 2007.

[11] «NI 9201 - National Instruments». [En línea]. Disponible en: http://sine.ni.com/nips/cds/view/p/lang/es/nid/208798. [Accedido: 18-mar-2017].

[12] «Cypress Unveils the World's Most Flexible One-Chip ARM Cortex-M0 Solution (NASDAQ:CY)». [En línea]. Disponible en: http://investors.cypress.com/releasedetail.cfm?releaseid=951486. [Accedido: 18-mar-2017].

[13] «System on a chip», *Wikipedia, la enciclopedia libre*. 03-mar-2016.

[14] «Integrated Automation System (IAS) - Høglund Marine Automation AS». [En línea]. Disponible en: /index.php?p=solutions/integrated-automation-system. [Accedido: 17-mar-2017].

[15] «ABB 800xA S800 I/O - ABB 800xA DCS Hardware - Controllers and I/O (ABB 800xA DCS distributed control system)». [En línea]. Disponible en: http://new.abb.com/control-systems/system-800xa/800xa-dcs/hardware-controllers-io/s800-i-o. [Accedido: 17-mar-2017].

[16] «Ingeteam Marine Automation». [En línea]. Disponible en: http://www.ingeteam.com/en-us/marine/automation/s17_34_p/products.aspx. [Accedido: 17-mar-2017].

[17] «Marine Automation System DIAMAR. Marine Systems», *Sedni*. .

[18] «Valmarine: Integrated Automation System (IAS)». [En línea]. Disponible en: http://www.valmarine.com/products2/automation-valmarine. [Accedido: 17-mar-2017].

[19] C. Falgueras, «Propuesta inicial de un sistema de monitorización de un bote de remo», Proyecto Final de Grado, Universidad de Sevilla, Sevilla, 2012.

[20] A. García Osés, «Diseño de una red CAN bus con Arduino», 2015.

[21] M. Á. G. Navarro, «Arduino based acquisition system for control applications», 2012.
[22] «Industruino Introduction», *Smarty*. [En línea]. Disponible en: http://widget.smartycenter.com/video/industruino-introduction/1454072/9050/1. [Accedido: 19-mar-2017].
[23] «Wiring». [En línea]. Disponible en: http://wiring.org.co/. [Accedido: 17-abr-2017].
[24] «Processing.org». [En línea]. Disponible en: https://processing.org/. [Accedido: 17-abr-2017].
[25] «Arduino - ArduinoBoardDue». [En línea]. Disponible en: https://www.arduino.cc/en/Main/arduinoBoardDue. [Accedido: 17-abr-2017].
[26] «home | Industruino». [En línea]. Disponible en: https://industruino.com/. [Accedido: 17-abr-2017].
[27] «Eclipse for C++ and Galileo Gen 2», *Intel® Galileo*. [En línea]. Disponible en: https://communities.intel.com/thread/108877. [Accedido: 24-abr-2017].
[28] «Yocto Project | Open Source embedded Linux build system, package metadata and SDK generator». [En línea]. Disponible en: https://www.yoctoproject.org/. [Accedido: 26-mar-2017].
[29] «Raspberry Pi 3 Model B», *Raspberry Pi*. .
[30] «Enable CANBus on the BeagleBone Black | Embedded Things». .
[31] «Beagleboard:BeagleBoneBlack - eLinux.org». [En línea]. Disponible en: http://elinux.org/Beagleboard:BeagleBoneBlack. [Accedido: 24-abr-2017].
[32] «embedded - Why do you need a Programmable Real Time Unit (PRU) while you can have an RTOS? - Stack Overflow». [En línea]. Disponible en: http://stackoverflow.com/questions/27092981/why-do-you-need-a-programmable-real-time-unit-pru-while-you-can-have-an-rtos. [Accedido: 24-abr-2017].
[33] «Samsung scraps a Raspberry Pi 3 competitor, shrinks Artik line», *PC World*. [En línea]. Disponible en: http://www.pcworld.idg.com.au/article/613824/samsung-scraps-raspberry-pi-3-competitor-shrinks-artik-line/. [Accedido: 25-abr-2017].
[34] «Códigos NRZ», *Wikipedia, la enciclopedia libre*. 27-mar-2017.
[35] «C Program for Bit Stuffing», *Get Program Code*, 08-mar-2013. .
[36] «CAN in Automation (CiA): Controller Area Network (CAN)». [En línea]. Disponible en: https://www.can-cia.org/. [Accedido: 01-may-2017].
[37] «can_physical_layer:main - CAN Wiki». [En línea]. Disponible en: http://www.can-wiki.info/doku.php?id=can_physical_layer:main. [Accedido: 01-may-2017].
[38] «Soubor:Numbered DE9 female Diagram.svg», *Wikipedie*. .
[39] «MODELO OSI VENTAJAS Y DESVENTAJAS». .
[40] «IEEE 802.3 ETHERNET». [En línea]. Disponible en: http://www.ieee802.org/3/. [Accedido: 05-may-2017].
[41] «Introduction to Ethernet, Fast Ethernet, Gigabit Ethernet». [En línea]. Disponible en: http://www.rhyshaden.com/eth_intr.htm. [Accedido: 05-may-2017].
[42] «What is CRC32 (Cyclic redundancy checksum)?» [En línea]. Disponible en: http://www.accuhash.com/what-is-crc32.html. [Accedido: 05-may-2017].

[43] «Data Communincation Standards and Protocols». [En línea]. Disponible en: http://www.eeherald.com/section/design-guide/ieee802_3.html. [Accedido: 06-may-2017].
[44] «Puerto de red - DB15 - MIDI , características y capacidades .:: www.informaticamoderna.com ::.» [En línea]. Disponible en: http://www.informaticamoderna.com/El_puerto_db15.htm. [Accedido: 06-may-2017].
[45] J. A. Tirado, «Sistemas Telemáticos: Thinnet y Thicknet», *Sistemas Telemáticos*, 25-sep-2010. .
[46] J. Black, *The System Engineers Handbook*. Elsevier, 2012.
[47] «Indeca - Industria del Cable». [En línea]. Disponible en: http://www.indeca.com.ar/productos/coaxiales-flexibles-de-50-y-75-ohms/18-RG-58-U. [Accedido: 06-may-2017].
[48] «Topologías Ethernet | Textos Científicos». [En línea]. Disponible en: https://www.textoscientificos.com/redes/ethernet/topologias-ethernet. [Accedido: 06-may-2017].
[49] A. J. I. Meza, «Instalación de Linux en una Intel Galileo», *Jorge Iván Meza Martínez*, 10-nov-2014. .
[50] «CAN-BUS Shield V1.2 - Shield for Arduino - Seeed Studio». [En línea]. Disponible en: https://www.seeedstudio.com/CAN-BUS-Shield-V1.2-p-2256.html. [Accedido: 25-mar-2017].
[51] «MCP2515 - Interface - Interface- Controller Area Network (CAN)». [En línea]. Disponible en: http://www.microchip.com/wwwproducts/en/en010406. [Accedido: 25-mar-2017].
[52] «MCP2551 - Interface - Interface- Controller Area Network (CAN)». [En línea]. Disponible en: http://www.microchip.com/wwwproducts/en/en010405. [Accedido: 25-mar-2017].
[53] «OBD», *Wikipedia, la enciclopedia libre*. 17-mar-2017.
[54] S. Team, «CAN-BUS Shield V1.2 - Seeed Wiki». [En línea]. Disponible en: http://wiki.seeed.cc/CAN-BUS_Shield_V1.2/. [Accedido: 25-mar-2017].
[55] «LM35DZ TEXAS INSTRUMENTS, Temperature Sensor IC, Voltage, ± 0.4°C, +2 °C, +100 °C, TO-92, 3 Pins | Farnell element14». [En línea]. Disponible en: http://uk.farnell.com/texas-instruments/lm35dz/ic-precision-temp-sensor-to-92/dp/9488200. [Accedido: 25-mar-2017].
[56] «LM35DZ/NOPB | Sensor de temperatura LM35DZ/NOPB, encapsulado TO-92 3 pines, interfaz Tensión, alimentación 4→ 30 V | Texas Instruments». [En línea]. Disponible en: http://es.rs-online.com/web/p/sensores-de-humedad-y-temperatura/5335907/. [Accedido: 25-mar-2017].
[57] «LM35 ±0.5°C Temperature Sensor with Analog Output with 30V Capability | TI.com». [En línea]. Disponible en: http://www.ti.com/product/LM35/datasheet/pin_configuration_and_functions#SNIS1593406. [Accedido: 25-mar-2017].
[58] «Fritzing». [En línea]. Disponible en: http://fritzing.org/. [Accedido: 28-mar-2017].
[59] «DHT11 basic temperature-humidity sensor + extras ID: 386 - $5.00 : Adafruit Industries, Unique & fun DIY electronics and kits». [En línea].

Disponible en: https://www.adafruit.com/product/386. [Accedido: 25-mar-2017].
[60] «Measuring Humidity with DHT11 Sensor», *GRobotronics Learning*, 09-jul-2013. .
[61] «gnu.org». [En línea]. Disponible en: https://www.gnu.org/philosophy/free-software-for-freedom.es.html. [Accedido: 26-mar-2017].
[62] «Intel® XDK | Intel® Software». [En línea]. Disponible en: https://software.intel.com/en-us/intel-xdk. [Accedido: 27-mar-2017].
[63] «Arduino IDE – Aplicaciones de Windows en Microsoft Store», *Microsoft Store*. [En línea]. Disponible en: https://www.microsoft.com/es-es/store/p/arduino-ide/9nblggh4rsd8. [Accedido: 27-mar-2017].
[64] «Download Intel® Galileo Firmware Updater and Drivers», *Drivers & Software*. [En línea]. Disponible en: https://downloadcenter.intel.com/download/26417/Intel-Galileo-Firmware-Updater-and-Drivers?v=t. [Accedido: 08-abr-2017].
[65] «IoT - Installing drivers and updating firmware for Arduino on a system with Windows* | Intel® Software». [En línea]. Disponible en: https://software.intel.com/en-us/installing-drivers-and-updating-firmware-for-arduino-windows. [Accedido: 08-abr-2017].
[66] «adafruit/DHT-sensor-library», *GitHub*. [En línea]. Disponible en: https://github.com/adafruit/DHT-sensor-library. [Accedido: 02-abr-2017].
[67] «DHT workaround for Galileo and Galileo Gen2 using 2 pins instead of one», *Intel® Galileo*. [En línea]. Disponible en: https://communities.intel.com/thread/53869. [Accedido: 17-mar-2017].
[68] «Arduino - Libraries». [En línea]. Disponible en: https://www.arduino.cc/en/Guide/Libraries#toc4. [Accedido: 02-abr-2017].
[69] «Seeed-Studio/CAN_BUS_Shield», *GitHub*. [En línea]. Disponible en: https://github.com/Seeed-Studio/CAN_BUS_Shield. [Accedido: 03-abr-2017].
[70] «coryjfowler/MCP_CAN_lib», *GitHub*. [En línea]. Disponible en: https://github.com/coryjfowler/MCP_CAN_lib. [Accedido: 03-abr-2017].
[71] C. Kim, «EECE456 Embedded Systems Desing LAB», Howard University.
[72] Luis, «Conectar Arduino a Internet o LAN con Shield Ethernet W5100», *Luis Llamas*. .
[73] designthemes, «El Shield Ethernet | Tutoriales Arduino». .
[74] «¿Qué es DHCP?» [En línea]. Disponible en: https://technet.microsoft.com/es-es/library/dd145320(v=ws.10).aspx. [Accedido: 08-abr-2017].
[75] «Arduino - FunctionDeclaration». [En línea]. Disponible en: https://www.arduino.cc/en/Reference/FunctionDeclaration. [Accedido: 13-abr-2017].
[76] «c++ - can a function return more than one value? - Stack Overflow». [En línea]. Disponible en: http://stackoverflow.com/questions/2571831/can-a-function-return-more-than-one-value. [Accedido: 13-abr-2017].
[77] F. J. Ceballos Sierra, *C/C++: curso de programación*. Paracuellos del Jarama, Madrid: RA-MA, 2015.

12. Anexos

12.1. Librería CAN Bus Shield

```
#include "mcp_can.h"

#define spi_readwrite SPI.transfer
#define spi_read() spi_readwrite(0x00)
#define SPI_BEGIN() SPI.beginTransaction(SPISettings(10000000,
MSBFIRST, SPI_MODE0))
#define SPI_END() SPI.endTransaction()

void MCP_CAN::mcp2515_reset(void)
{
#ifdef SPI_HAS_TRANSACTION
    SPI_BEGIN();
#endif
    MCP2515_SELECT();
    spi_readwrite(MCP_RESET);
    MCP2515_UNSELECT();
#ifdef SPI_HAS_TRANSACTION
    SPI_END();
#endif
    delay(10);
}

byte MCP_CAN::mcp2515_readRegister(const byte address)
{
    byte ret;

#ifdef SPI_HAS_TRANSACTION
    SPI_BEGIN();
#endif
    MCP2515_SELECT();
    spi_readwrite(MCP_READ);
    spi_readwrite(address);
    ret = spi_read();
    MCP2515_UNSELECT();
#ifdef SPI_HAS_TRANSACTION
    SPI_END();
#endif

    return ret;
}

void MCP_CAN::mcp2515_readRegisterS(const byte address, byte values[],
const byte n)
{
    byte i;
#ifdef SPI_HAS_TRANSACTION
    SPI_BEGIN();
#endif
    MCP2515_SELECT();
    spi_readwrite(MCP_READ);
    spi_readwrite(address);
    // mcp2515 has auto-increment of address-pointer
```

```
    for(i=0; i<n && i<CAN_MAX_CHAR_IN_MESSAGE; i++) {
        values[i] = spi_read();
    }
    MCP2515_UNSELECT();
#ifdef SPI_HAS_TRANSACTION
    SPI_END();
#endif
}
void MCP_CAN::mcp2515_setRegister(const byte address, const byte
value)
{
#ifdef SPI_HAS_TRANSACTION
    SPI_BEGIN();
#endif
    MCP2515_SELECT();
    spi_readwrite(MCP_WRITE);
    spi_readwrite(address);
    spi_readwrite(value);
    MCP2515_UNSELECT();
#ifdef SPI_HAS_TRANSACTION
    SPI_END();
#endif
}

void MCP_CAN::mcp2515_setRegisterS(const byte address, const byte
values[], const byte n)
{
    byte i;
#ifdef SPI_HAS_TRANSACTION
    SPI_BEGIN();
#endif
    MCP2515_SELECT();
    spi_readwrite(MCP_WRITE);
    spi_readwrite(address);

    for(i=0; i<n; i++)
    {
        spi_readwrite(values[i]);
    }
    MCP2515_UNSELECT();
#ifdef SPI_HAS_TRANSACTION
    SPI_END();
#endif
}
void MCP_CAN::mcp2515_modifyRegister(const byte address, const byte
mask, const byte data)
{
#ifdef SPI_HAS_TRANSACTION
    SPI_BEGIN();
#endif
    MCP2515_SELECT();
    spi_readwrite(MCP_BITMOD);
    spi_readwrite(address);
    spi_readwrite(mask);
    spi_readwrite(data);
    MCP2515_UNSELECT();
#ifdef SPI_HAS_TRANSACTION
    SPI_END();
#endif
}
byte MCP_CAN::mcp2515_readStatus(void)
```

```
{
    byte i;
#ifdef SPI_HAS_TRANSACTION
    SPI_BEGIN();
#endif
    MCP2515_SELECT();
    spi_readwrite(MCP_READ_STATUS);
    i = spi_read();
    MCP2515_UNSELECT();
#ifdef SPI_HAS_TRANSACTION
    SPI_END();
#endif

    return i;
}

byte MCP_CAN::mcp2515_setCANCTRL_Mode(const byte newmode)
{
    byte i;

    mcp2515_modifyRegister(MCP_CANCTRL, MODE_MASK, newmode);

    i = mcp2515_readRegister(MCP_CANCTRL);
    i &= MODE_MASK;

    if(i == newmode )
    {
        return MCP2515_OK;
    }

    return MCP2515_FAIL;
}

byte MCP_CAN::mcp2515_configRate(const byte canSpeed)
{
    byte set, cfg1, cfg2, cfg3;
    set = 1;
    switch (canSpeed)
    {
        case (CAN_5KBPS):
        cfg1 = MCP_16MHz_5kBPS_CFG1;
        cfg2 = MCP_16MHz_5kBPS_CFG2;
        cfg3 = MCP_16MHz_5kBPS_CFG3;
        break;

        case (CAN_10KBPS):
        cfg1 = MCP_16MHz_10kBPS_CFG1;
        cfg2 = MCP_16MHz_10kBPS_CFG2;
        cfg3 = MCP_16MHz_10kBPS_CFG3;
        break;

        case (CAN_20KBPS):
        cfg1 = MCP_16MHz_20kBPS_CFG1;
        cfg2 = MCP_16MHz_20kBPS_CFG2;
        cfg3 = MCP_16MHz_20kBPS_CFG3;
        break;

        case (CAN_25KBPS):
        cfg1 = MCP_16MHz_25kBPS_CFG1;
        cfg2 = MCP_16MHz_25kBPS_CFG2;
        cfg3 = MCP_16MHz_25kBPS_CFG3;
```

```
break;

case (CAN_31K25BPS):
cfg1 = MCP_16MHz_31k25BPS_CFG1;
cfg2 = MCP_16MHz_31k25BPS_CFG2;
cfg3 = MCP_16MHz_31k25BPS_CFG3;
break;

case (CAN_33KBPS):
cfg1 = MCP_16MHz_33kBPS_CFG1;
cfg2 = MCP_16MHz_33kBPS_CFG2;
cfg3 = MCP_16MHz_33kBPS_CFG3;
break;

case (CAN_40KBPS):
cfg1 = MCP_16MHz_40kBPS_CFG1;
cfg2 = MCP_16MHz_40kBPS_CFG2;
cfg3 = MCP_16MHz_40kBPS_CFG3;
break;

case (CAN_50KBPS):
cfg1 = MCP_16MHz_50kBPS_CFG1;
cfg2 = MCP_16MHz_50kBPS_CFG2;
cfg3 = MCP_16MHz_50kBPS_CFG3;
break;

case (CAN_80KBPS):
cfg1 = MCP_16MHz_80kBPS_CFG1;
cfg2 = MCP_16MHz_80kBPS_CFG2;
cfg3 = MCP_16MHz_80kBPS_CFG3;
break;

case (CAN_83K3BPS):
cfg1 = MCP_16MHz_83k3BPS_CFG1;
cfg2 = MCP_16MHz_83k3BPS_CFG2;
cfg3 = MCP_16MHz_83k3BPS_CFG3;
break;

case (CAN_95KBPS):
cfg1 = MCP_16MHz_95kBPS_CFG1;
cfg2 = MCP_16MHz_95kBPS_CFG2;
cfg3 = MCP_16MHz_95kBPS_CFG3;
break;

case (CAN_100KBPS):
cfg1 = MCP_16MHz_100kBPS_CFG1;
cfg2 = MCP_16MHz_100kBPS_CFG2;
cfg3 = MCP_16MHz_100kBPS_CFG3;
break;

case (CAN_125KBPS):
cfg1 = MCP_16MHz_125kBPS_CFG1;
cfg2 = MCP_16MHz_125kBPS_CFG2;
cfg3 = MCP_16MHz_125kBPS_CFG3;
break;

case (CAN_200KBPS):
cfg1 = MCP_16MHz_200kBPS_CFG1;
cfg2 = MCP_16MHz_200kBPS_CFG2;
cfg3 = MCP_16MHz_200kBPS_CFG3;
break;
```

```
        case (CAN_250KBPS):
        cfg1 = MCP_16MHz_250kBPS_CFG1;
        cfg2 = MCP_16MHz_250kBPS_CFG2;
        cfg3 = MCP_16MHz_250kBPS_CFG3;
        break;

        case (CAN_500KBPS):
        cfg1 = MCP_16MHz_500kBPS_CFG1;
        cfg2 = MCP_16MHz_500kBPS_CFG2;
        cfg3 = MCP_16MHz_500kBPS_CFG3;
        break;

        case (CAN_666KBPS):
        cfg1 = MCP_16MHz_666kBPS_CFG1;
        cfg2 = MCP_16MHz_666kBPS_CFG2;
        cfg3 = MCP_16MHz_666kBPS_CFG3;
        break;

        case (CAN_1000KBPS):
        cfg1 = MCP_16MHz_1000kBPS_CFG1;
        cfg2 = MCP_16MHz_1000kBPS_CFG2;
        cfg3 = MCP_16MHz_1000kBPS_CFG3;
        break;

        default:
        set = 0;
        break;
    }

    if(set) {
        mcp2515_setRegister(MCP_CNF1, cfg1);
        mcp2515_setRegister(MCP_CNF2, cfg2);
        mcp2515_setRegister(MCP_CNF3, cfg3);
        return MCP2515_OK;
    }
    else {
        return MCP2515_FAIL;
    }
}

void MCP_CAN::mcp2515_initCANBuffers(void)
{
    byte i, a1, a2, a3;

    a1 = MCP_TXB0CTRL;
    a2 = MCP_TXB1CTRL;
    a3 = MCP_TXB2CTRL;
    for(i = 0; i < 14; i++)
{                                           // in-buffer loop
        mcp2515_setRegister(a1, 0);
        mcp2515_setRegister(a2, 0);
        mcp2515_setRegister(a3, 0);
        a1++;
        a2++;
        a3++;
    }
    mcp2515_setRegister(MCP_RXB0CTRL, 0);
    mcp2515_setRegister(MCP_RXB1CTRL, 0);
}
byte MCP_CAN::mcp2515_init(const byte canSpeed)
```

```
{

    byte res;

    mcp2515_reset();

    res = mcp2515_setCANCTRL_Mode(MODE_CONFIG);
    if(res > 0)
    {
#if DEBUG_EN
        Serial.print("Enter setting mode fall\r\n");
#else
        delay(10);
#endif
        return res;
    }
#if DEBUG_EN
    Serial.print("Enter setting mode success \r\n");
#else
    delay(10);
#endif

    // set boadrate
    if(mcp2515_configRate(canSpeed))
    {
#if DEBUG_EN
        Serial.print("set rate fall!!\r\n");
#else
        delay(10);
#endif
        return res;
    }
#if DEBUG_EN
    Serial.print("set rate success!!\r\n");
#else
    delay(10);
#endif

    if(res == MCP2515_OK ) {

        // init canbuffers
        mcp2515_initCANBuffers();

        // interrupt mode
        mcp2515_setRegister(MCP_CANINTE, MCP_RX0IF | MCP_RX1IF);

#if (DEBUG_RXANY==1)
        // enable both receive-buffers to receive any message and
enable rollover
        mcp2515_modifyRegister(MCP_RXB0CTRL,
        MCP_RXB_RX_MASK | MCP_RXB_BUKT_MASK,
        MCP_RXB_RX_ANY | MCP_RXB_BUKT_MASK);
        mcp2515_modifyRegister(MCP_RXB1CTRL, MCP_RXB_RX_MASK,
        MCP_RXB_RX_ANY);
#else
        // enable both receive-buffers to receive messages with std.
and ext. identifiers and enable rollover
        mcp2515_modifyRegister(MCP_RXB0CTRL,
        MCP_RXB_RX_MASK | MCP_RXB_BUKT_MASK,
        MCP_RXB_RX_STDEXT | MCP_RXB_BUKT_MASK );
        mcp2515_modifyRegister(MCP_RXB1CTRL, MCP_RXB_RX_MASK,
```

```
        MCP_RXB_RX_STDEXT);
#endif
        // enter normal mode
        res = mcp2515_setCANCTRL_Mode(MODE_NORMAL);
        if(res)
        {
#if DEBUG_EN
            Serial.print("Enter Normal Mode Fall!!\r\n");
#else
            delay(10);
#endif
            return res;
        }


#if DEBUG_EN
        Serial.print("Enter Normal Mode Success!!\r\n");
#else
        delay(10);
#endif

    }
    return res;

}

void MCP_CAN::mcp2515_write_id( const byte mcp_addr, const byte ext,
const unsigned long id )
{
    uint16_t canid;
    byte tbufdata[4];

    canid = (uint16_t)(id & 0x0FFFF);

    if(ext == 1)
    {
        tbufdata[MCP_EID0] = (byte) (canid & 0xFF);
        tbufdata[MCP_EID8] = (byte) (canid >> 8);
        canid = (uint16_t)(id >> 16);
        tbufdata[MCP_SIDL] = (byte) (canid & 0x03);
        tbufdata[MCP_SIDL] += (byte) ((canid & 0x1C) << 3);
        tbufdata[MCP_SIDL] |= MCP_TXB_EXIDE_M;
        tbufdata[MCP_SIDH] = (byte) (canid >> 5 );
    }
    else
    {
        tbufdata[MCP_SIDH] = (byte) (canid >> 3 );
        tbufdata[MCP_SIDL] = (byte) ((canid & 0x07 ) << 5);
        tbufdata[MCP_EID0] = 0;
        tbufdata[MCP_EID8] = 0;
    }
    mcp2515_setRegisterS( mcp_addr, tbufdata, 4 );
}

void MCP_CAN::mcp2515_read_id( const byte mcp_addr, byte* ext,
unsigned long* id )
{
    byte tbufdata[4];

    *ext = 0;
    *id = 0;
```

```
    mcp2515_readRegisterS( mcp_addr, tbufdata, 4 );

    *id = (tbufdata[MCP_SIDH]<<3) + (tbufdata[MCP_SIDL]>>5);

    if((tbufdata[MCP_SIDL] & MCP_TXB_EXIDE_M) ==  MCP_TXB_EXIDE_M )
    {
        // extended id
        *id = (*id<<2) + (tbufdata[MCP_SIDL] & 0x03);
        *id = (*id<<8) + tbufdata[MCP_EID8];
        *id = (*id<<8) + tbufdata[MCP_EID0];
        *ext = 1;
    }
}
void MCP_CAN::mcp2515_write_canMsg( const byte buffer_sidh_addr)
{
    byte mcp_addr;
    mcp_addr = buffer_sidh_addr;
    mcp2515_setRegisterS(mcp_addr+5, dta, dta_len );
// write data bytes
    if(rtr == 1)                                                    //
if RTR set bit in byte
    {
        dta_len |= MCP_RTR_MASK;
    }
    mcp2515_setRegister((mcp_addr+4), dta_len );
// write the RTR and DLC
    mcp2515_write_id(mcp_addr, ext_flg, can_id );
// write CAN id

}

void MCP_CAN::mcp2515_read_canMsg( const byte buffer_sidh_addr)
// read can msg
{
    byte mcp_addr, ctrl;

    mcp_addr = buffer_sidh_addr;
    mcp2515_read_id( mcp_addr, &ext_flg,&can_id );
    ctrl = mcp2515_readRegister( mcp_addr-1 );
    dta_len = mcp2515_readRegister( mcp_addr+4 );

    if((ctrl & 0x08)) {
        rtr = 1;
    }
    else {
        rtr = 0;
    }

    dta_len &= MCP_DLC_MASK;
    mcp2515_readRegisterS( mcp_addr+5, &(dta[0]), dta_len );
}

void MCP_CAN::mcp2515_start_transmit(const byte mcp_addr)
// start transmit
{
    mcp2515_modifyRegister( mcp_addr-1 , MCP_TXB_TXREQ_M,
MCP_TXB_TXREQ_M );
}
```

```
byte MCP_CAN::mcp2515_getNextFreeTXBuf(byte *txbuf_n)
// get Next free txbuf
{
    byte res, i, ctrlval;
    byte ctrlregs[MCP_N_TXBUFFERS] = { MCP_TXB0CTRL, MCP_TXB1CTRL,
MCP_TXB2CTRL };

    res = MCP_ALLTXBUSY;
    *txbuf_n = 0x00;

    // check all 3 TX-Buffers
    for(i=0; i<MCP_N_TXBUFFERS; i++) {
        ctrlval = mcp2515_readRegister( ctrlregs[i] );
        if((ctrlval & MCP_TXB_TXREQ_M) == 0 ) {
            *txbuf_n = ctrlregs[i]+1;
// return SIDH-address of Buffe
            // r
            res = MCP2515_OK;
            return res;
// ! function exit
        }
    }
    return res;
}
MCP_CAN::MCP_CAN(byte _CS)
{
    SPICS = _CS;
    pinMode(SPICS, OUTPUT);
    MCP2515_UNSELECT();
}

byte MCP_CAN::begin(byte speedset)
{
    byte res;

    SPI.begin();
    res = mcp2515_init(speedset);
    if(res == MCP2515_OK) return CAN_OK;
    else return CAN_FAILINIT;
}


byte MCP_CAN::init_Mask(byte num, byte ext, unsigned long ulData)
{
    byte res = MCP2515_OK;
#if DEBUG_EN
    Serial.print("Begin to set Mask!!\r\n");
#else
    delay(10);
#endif
    res = mcp2515_setCANCTRL_Mode(MODE_CONFIG);
    if(res > 0){
#if DEBUG_EN
        Serial.print("Enter setting mode fall\r\n");
#else
        delay(10);
#endif
        return res;
    }

    if(num == 0){
```

```
        mcp2515_write_id(MCP_RXM0SIDH, ext, ulData);

    }
    else if(num == 1){
        mcp2515_write_id(MCP_RXM1SIDH, ext, ulData);
    }
    else res =  MCP2515_FAIL;

    res = mcp2515_setCANCTRL_Mode(MODE_NORMAL);
    if(res > 0){
#if DEBUG_EN
        Serial.print("Enter normal mode fall\r\n");
#else
        delay(10);
#endif
        return res;
    }
#if DEBUG_EN
    Serial.print("set Mask success!!\r\n");
#else
    delay(10);
#endif
    return res;
}
byte MCP_CAN::init_Filt(byte num, byte ext, unsigned long ulData)
{
    byte res = MCP2515_OK;
#if DEBUG_EN
    Serial.print("Begin to set Filter!!\r\n");
#else
    delay(10);
#endif
    res = mcp2515_setCANCTRL_Mode(MODE_CONFIG);
    if(res > 0)
    {
#if DEBUG_EN
        Serial.print("Enter setting mode fall\r\n");
#else
        delay(10);
#endif
        return res;
    }

    switch( num )
    {
        case 0:
        mcp2515_write_id(MCP_RXF0SIDH, ext, ulData);
        break;

        case 1:
        mcp2515_write_id(MCP_RXF1SIDH, ext, ulData);
        break;

        case 2:
        mcp2515_write_id(MCP_RXF2SIDH, ext, ulData);
        break;

        case 3:
        mcp2515_write_id(MCP_RXF3SIDH, ext, ulData);
        break;
```

```
            case 4:
            mcp2515_write_id(MCP_RXF4SIDH, ext, ulData);
            break;

            case 5:
            mcp2515_write_id(MCP_RXF5SIDH, ext, ulData);
            break;

            default:
            res = MCP2515_FAIL;
        }

        res = mcp2515_setCANCTRL_Mode(MODE_NORMAL);
        if(res > 0)
        {
#if DEBUG_EN
            Serial.print("Enter normal mode fall\r\nSet filter
fail!!\r\n");
#else
            delay(10);
#endif
            return res;
        }
#if DEBUG_EN
        Serial.print("set Filter success!!\r\n");
#else
        delay(10);
#endif

        return res;
}

byte MCP_CAN::setMsg(unsigned long id, byte ext, byte len, byte rtr,
byte *pData)
{
    ext_flg = ext;
    can_id     = id;
    dta_len    = min( len, MAX_CHAR_IN_MESSAGE );
    rtr    = rtr;
    for(int i = 0; i<dta_len; i++)
    {
        dta[i] = *(pData+i);
    }
    return MCP2515_OK;
}
byte MCP_CAN::setMsg(unsigned long id, byte ext, byte len, byte
*pData)
{
    return setMsg( id, ext, len, 0, pData );
}
byte MCP_CAN::clearMsg()
{
    can_id       = 0;
    dta_len      = 0;
    ext_flg   = 0;
    rtr      = 0;
    filhit   = 0;
    for(int i = 0; i<dta_len; i++ )
    dta[i] = 0x00;

    return MCP2515_OK;
```

```
}

byte MCP_CAN::sendMsg()
{
    byte res, res1, txbuf_n;
    uint16_t uiTimeOut = 0;

    do {
        res = mcp2515_getNextFreeTXBuf(&txbuf_n);
// info = addr.
        uiTimeOut++;
    } while (res == MCP_ALLTXBUSY && (uiTimeOut < TIMEOUTVALUE));

    if(uiTimeOut == TIMEOUTVALUE)
    {
        return CAN_GETTXBFTIMEOUT;
// get tx buff time out
    }
    uiTimeOut = 0;
    mcp2515_write_canMsg( txbuf_n);
    mcp2515_start_transmit( txbuf_n );
    do
    {
        uiTimeOut++;
        res1= mcp2515_readRegister(txbuf_n-1 /* the ctrl reg is
located at txbuf_n-1 */);  // read send buff ctrl reg
        res1 = res1 & 0x08;
    }while(res1 && (uiTimeOut < TIMEOUTVALUE));
    if(uiTimeOut == TIMEOUTVALUE)
// send msg timeout
    {
        return CAN_SENDMSGTIMEOUT;
    }
    return CAN_OK;

}
byte MCP_CAN::sendMsgBuf(unsigned long id, byte ext, byte rtr, byte
len, byte *buf)
{
    setMsg(id, ext, len, rtr, buf);
    return sendMsg();
}
byte MCP_CAN::sendMsgBuf(unsigned long id, byte ext, byte len, byte
*buf)
{
    setMsg(id, ext, len, buf);
    return sendMsg();
}

byte MCP_CAN::readMsg()
{
    byte stat, res;

    stat = mcp2515_readStatus();

    if(stat & MCP_STAT_RX0IF )
// Msg in Buffer 0
    {
        mcp2515_read_canMsg( MCP_RXBUF_0);
        mcp2515_modifyRegister(MCP_CANINTF, MCP_RX0IF, 0);
        res = CAN_OK;
```

```
    }
    else if(stat & MCP_STAT_RX1IF )
// Msg in Buffer 1
    {
        mcp2515_read_canMsg( MCP_RXBUF_1);
        mcp2515_modifyRegister(MCP_CANINTF, MCP_RX1IF, 0);
        res = CAN_OK;
    }
    else
    {
        res = CAN_NOMSG;
    }
    return res;
}

byte MCP_CAN::readMsgBuf(byte *len, byte buf[])
{
    byte  rc;

    rc = readMsg();

    if(rc == CAN_OK) {
        *len = dta_len;
        for(int i = 0; i<dta_len; i++) {
            buf[i] = dta[i];
        }
    } else {
        *len = 0;
    }
    return rc;
}
byte MCP_CAN::readMsgBufID(unsigned long *ID, byte *len, byte buf[])
{
    byte rc;
    rc = readMsg();

    if(rc == CAN_OK) {
        *len = dta_len;
        *ID  = can_id;
        for(int i = 0; i<dta_len && i < MAX_CHAR_IN_MESSAGE; i++) {
            buf[i] = dta[i];
        }
    } else {
        *len = 0;
    }
    return rc;
}

byte MCP_CAN::checkReceive(void)
{
    byte res;
    res = mcp2515_readStatus();
// RXnIF in Bit 1 and 0
    if(res & MCP_STAT_RXIF_MASK )
    {
        return CAN_MSGAVAIL;
    }
    else
    {
        return CAN_NOMSG;
    }
```

```
}

byte MCP_CAN::checkError(void)
{
    byte eflg = mcp2515_readRegister(MCP_EFLG);

    if(eflg & MCP_EFLG_ERRORMASK )
    {
        return CAN_CTRLERROR;
    }
    else
    {
        return CAN_OK;
    }
}
unsigned long MCP_CAN::getCanId(void)
{
    return can_id;
}
byte MCP_CAN::isRemoteRequest(void)
{
    return rtr;
}

byte MCP_CAN::isExtendedFrame(void)
{
    return ext_flg;
}
```

12.2. Librería DHT

```
#include "DHT.h"

DHT::DHT(uint8_t inPin, uint8_t outPin, uint8_t type, uint8_t count) {
  _inpin = inPin;
  _outpin = outPin;
  _type = type;
  _count = count;
  firstreading = true;
}

void DHT::begin(void) {
  pinMode(_inpin, INPUT);
  pinMode(_outpin, INPUT);
  digitalWrite(_outpin, HIGH);
  _lastreadtime = 0;
}

float DHT::readTemperature(bool S) {
  float f;

  if (read()) {
    switch (_type) {
    case DHT11:
      f = data[2];
      if(S)
        f = convertCtoF(f);

      return f;
```

```
    case DHT22:
    case DHT21:
      f = data[2] & 0x7F;
      f *= 256;
      f += data[3];
      f /= 10;
      if (data[2] & 0x80)
    f *= -1;
      if(S)
    f = convertCtoF(f);

      return f;
    }
  }
  return NAN;
}

float DHT::convertCtoF(float c) {
    return c * 9 / 5 + 32;
}

float DHT::readHumidity(void) {
  float f;
  if (read()) {
    switch (_type) {
    case DHT11:
      f = data[0];
      return f;
    case DHT22:
    case DHT21:
      f = data[0];
      f *= 256;
      f += data[1];
      f /= 10;
      return f;
    }
  }
  return NAN;
}

float DHT::computeHeatIndex(float tempFahrenheit, float
percentHumidity) {

  return -42.379 +
           2.04901523 * tempFahrenheit +
          10.14333127 * percentHumidity +
          -0.22475541 * tempFahrenheit*percentHumidity +
          -0.00683783 * pow(tempFahrenheit, 2) +
          -0.05481717 * pow(percentHumidity, 2) +
           0.00122874 * pow(tempFahrenheit, 2) * percentHumidity +
           0.00085282 * tempFahrenheit*pow(percentHumidity, 2) +
          -0.00000199 * pow(tempFahrenheit, 2) * pow(percentHumidity,
2);
}

boolean DHT::read(void) {
  uint8_t laststate = HIGH;
  uint8_t counter = 0;
  uint8_t j = 0, i;
  unsigned long currenttime;
```

```
  int bitContainer[8];
  currenttime = millis();
  if (currenttime < _lastreadtime) {
    _lastreadtime = 0;
  }
  if (!firstreading && ((currenttime - _lastreadtime) < 2000)) {
    return true; // return last correct measurement
  }
  firstreading = false;
  _lastreadtime = millis();

  data[0] = data[1] = data[2] = data[3] = data[4] = 0;
  pinMode(_outpin, OUTPUT_FAST);
  pinMode(_inpin, INPUT_FAST);
  digitalWrite(_outpin, HIGH);
  delay(250);
  noInterrupts();
  digitalWrite(_outpin, LOW);
  delay(20);
  digitalWrite(_outpin, HIGH);
  delayMicrosGal(40);
  delayMicrosGal(160);
  for(int bytes = 0; bytes < 5; bytes++)
  {
        for(int i = 0; i < 8; i++)
        {
            int pulse = pulseLength(_inpin);
            if(pulse > 30)
            {
                  bitContainer[i] = 1;
            }
            else
            {
                  bitContainer[i] = 0;
            }
        }
        data[bytes] = bitsToByte(bitContainer);
  }

  interrupts();

    if((data[4] == ((data[0] + data[1] + data[2] + data[3]) & 0xFF)))
      {
            return true;
      }

  return false;

}

void DHT::delayMicrosGal(unsigned long usec)
{
    unsigned long a = micros();
    unsigned long b = a;
    while((b-a) < usec)
    {
        b = micros();
    }
}

int DHT::pulseLength(int pin)
```

```
{
    unsigned long a = micros();
    unsigned long b = a;
    unsigned long c = a;
    int timeout = 150;
      int fastPin = 0;
    int highValue = 0;

    if(PLATFORM_NAME == "GalileoGen2")
    {
      switch(pin)
      {
      case 0:
          fastPin = GPIO_FAST_ID_QUARK_SC(0x08);
        highValue = 0x08;
        break;
      case 1:
        fastPin = GPIO_FAST_ID_QUARK_SC(0x10);
        highValue = 0x10;
        break;
      case 2:
        fastPin = GPIO_FAST_ID_QUARK_SC(0x20);
        highValue = 0x20;
        break;
      case 3:
        fastPin = GPIO_FAST_ID_QUARK_SC(0x40);
        highValue = 0x40;
        break;
      case 4:
        fastPin = GPIO_FAST_ID_QUARK_NC_RW(0x10);
        highValue = 0x10;
        break;
      case 5:
        fastPin = GPIO_FAST_ID_QUARK_NC_CW(0x01);
        highValue = 0x01;
        break;
      case 6:
        fastPin = GPIO_FAST_ID_QUARK_NC_CW(0x02);
        highValue = 0x02;
        break;
      case 9:
        fastPin = GPIO_FAST_ID_QUARK_NC_RW(0x04);
        highValue = 0x04;
        break;
      case 10:
        fastPin = GPIO_FAST_ID_QUARK_SC(0x04);
        highValue = 0x04;
        break;
      case 11:
        fastPin = GPIO_FAST_ID_QUARK_NC_RW(0x08);
        highValue = 0x08;
        break;
      case 12:
        fastPin = GPIO_FAST_ID_QUARK_SC(0x80);
        highValue = 0x80;
        break;
      case 13:
        fastPin = GPIO_FAST_ID_QUARK_NC_RW(0x20);
        highValue = 0x20;
        break;
      default:
```

```
        highValue = 1;
        break;
      }
  }
  else
  {
    switch(_inpin)
    {
    case 2:
      fastPin = GPIO_FAST_ID_QUARK_SC(0x40);
      highValue = 0x40;
      break;
    case 3:
      fastPin = GPIO_FAST_ID_QUARK_SC(0x80);
      highValue = 0x80;
      break;
    default:
      highValue = 1;
      break;
    }
  }
  a = micros();
  while(fastGpioDigitalRead(fastPin) == 0)
  {
    b= micros();
    if((b - a) >= timeout)
    {
        break;
    }
  }
  a = micros();
  while(fastGpioDigitalRead(fastPin) == highValue)
  {
      b= micros();
    if((b - a) >= timeout)
    {
        break;
    }
  }
  return (b - a);
  return 0;
}
int DHT::bitsToByte(int bits[])
{
      int data = 0;
      for(int i = 0; i < 8; i++)
      {
            if (bits[i])
            {
            data |= (int)(1 << (7 - i));
            }
      }
      return data;
}
```