Proyecto Fin de Carrera
Ingeniería de las Tecnologías Industriales
Mención en Electrónica

Diseño y desarrollo de un sistema de refrigeración distribuido basado en comunicaciones CANBUS

Autor: Rafael Romero García

Tutor: Juan Antonio Sánchez Segura

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Proyecto Fin de Carrera
Ingeniería de las Tecnologías Industriales

Diseño y desarrollo de un sistema de refrigeración distribuido basado en comunicaciones CANBUS

Autor:
Rafael Romero García

Tutor:
Juan Antonio Sánchez Segura
Profesor colaborador

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Proyecto Fin de Carrera: Diseño y desarrollo de un sistema de refrigeración distribuido basado en comunicaciones CANBUS

Autor: Rafael Romero García

Tutor: Juan Antonio Sánchez Segura

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

Agradecimientos

A mi familia, novia y amigos por el apoyo incondicional; a Juan Antonio por guiarme; y a mis compañeros de piso por la ayuda. Gracias a todos.

Rafael Romero García

Sevilla, 2020

Resumen

Este proyecto se ha realizado sobre el LaunchPad Tiva™ C Series TM4C123G de Texas Instruments y el transceptor CANbus MCP2562 de Microchip. Presenta un doble objetivo: primero, el desarrollo de un sistema de control de refrigeración distribuido mediante el uso de ventiladores y sensores de temperatura conectados a launchpads; segundo, la expansión de las posibilidades en relación con el launchpad y las prácticas impartidas en el M.I.E.R.A, dejando la posibilidad de expandir tanto el proyecto como las prácticas.

El trabajo cuenta con dos partes bien diferenciadas:

La primera parte consiste en la realización, mediante el programa informático "KiCad" (software libre similar al "Eagle"), de una placa similar al boosterpack utilizado en las prácticas del Master en Ingeniería Electrónica, Robótica y Automática (M.I.E.R.A) que permita incluir la conexión de los ventiladores y los sensores de temperatura.

La segunda parte consiste en la programación por medio del programa informático "Code Composer Studio" (CCS) de tres terminales: dos que actúan como nodos, que se van a encargar de leer la temperatura por medio de sensores y de accionar los ventiladores; y uno que actúa como master, recibiendo la temperatura y convirtiéndola a un valor de pwm para cada ventilador.

Abstract

This project has been done with the Texas Intruments Tiva™ C Series TM4C123G LaunchPad and the Microchip™ CANbus MCP2562 transceptor. It has two objectives: first, the development of a distributed refrigeration control system using fans and temperature sesnsors connected to the launchpads; and second, the expansion of the possibilities related to the launchpad and the practices taught in the "Master en Ingeniería Electrónica, Robótica y Automática" (M.I.E.R.A.), leaving the possibility of expanding both the project and the practices.

The project has two main parts:

The first part, using the software "KiCad" (a free software similar to "Eagle"), is the making of a board similar to the boosterpack used in the practices that allows to include the fans and the temperature sensors.

The second part is the making of the code, using "Code Composer Studio" (CCS), of the three terminals: two of them working as the nodes that are going to read the temperature and activate the fans; and one working as the master, which is going to receive the temperature and convert it to a pwm value for each fan.

ÍNDICE

ÍNDICE DE FIGURAS

Notación

ALU	Arithmetic Logic Unit
CAN	Controller Area Network
CCS	Code Composer Studio
CPU	Central Processing Unit
GPIO	General Purpose Input-Output
M.I.E.R.A.	Master en Ingeniería Electrónica, Robótica y Automática
PCB	Printed Circuit Board
PWM	Pulse Width Modulation
MOS	Metal oxide semiconductor
NVIC	Nested Vector Interrupt Control
RAM	Random Access Memory
ROM	Read Only Memory
UC	Unidad de Control
USB	Universal Serial Bus

1 INTRODUCCIÓN

1.1 Planteamiento

1.1.1 Idea de proyecto

Desde un primer momento se tuvo la idea de realizar un proyecto relacionado con algún tipo de comunicación mediante el uso de microcontroladores. Un proyecto que ayudase a trabajar y conocer mejor tanto el aspecto de la programación de estos dispositivos como el hardware. Se propuso el control del sistema de refrigeración del interior de un coche y es por eso que se optó por este trabajo. Aunque el fin que se busca se puede lograr con distintos dispositivos, como una Raspberry Pi o una placa de Arduino, el departamento de electrónica ya poseía las placas Tiva, luego se podía trabajar con un microcontrolador que contaba con el bus CAN, (que además es el sistema por excelencia en los vehículos) sin necesidad de aumentar el presupuesto del trabajo con la compra de la placa.

Finalmente, solo se llegó a la configuración del sistema distribuido con varios nodos, que simularían distintas zonas del coche, sin entrar en otros aspectos, como la integración del sistema dentro de la aparamenta del vehículo u otros factores exteriores a este.

1.1.2 Objetivos

- Objetivo principal: Desarrollo de un sistema de refrigeración distribuido mediante ventiladores que respondan a información proporcionada por sensores de temperatura.
- Objetivos específicos:
  - Desarrollo y fabricación de una placa que permita implementar el CAN, los sensores de temperatura y los ventiladores.
  - Programación de la comunicación entre los nodos basada en el protocolo CAN.
- Objetivos generales:
  - Familiarizarse con el trabajo con un microcontrolador y con la comunicación con otros dispositivos por medio de los periféricos dedicados a ello.
  - Expansión de las prácticas del primer curso de Comunicaciones Industriales del M.I.E.R.A.

1.2 Microcontroladores

1.2.1 Historia

En el año 1971, Intel fabricó el primer microprocesador, el 4004, de tecnología PMOS. Era un microprocesador de 4 bits que ejecutaba una instrucción cada 20 microsegundos de promedio y solo podía direccionar 4k localidades de memoria. En los siguientes años, Intel mejoró su creación, lanzando el 8008 (microprocesador de 8 bits, aunque igual de limitado y lento en la ejecución de instrucciones) y el 8080 (de tecnología NMOS, más rápido que sus antecesores). Tras esto, se produjo un auge en el diseño de microcomputadores, sumándose nuevas empresas como Motorola o National Semiconductors.

No fue hasta el año 1974 que Texas Instruments comenzó a comercializar el primer microcontrolador, el TMS1000, que contaba en un solo chip con memoria RAM, ROM, un microprocesador y un reloj interno. Fue desarrollado a partir del modelo anterior (el TMS0100) por los ingenieros de T.I. Gary Boone y Michael Cochran, en 1971. En poco tiempo se sumarían otras empresas, principalmente Intel y Motorola.

En la década de los 80 la tecnología avanzó tanto para los microprocesadores como para los microcontroladores, los primeros destinados cada vez más a trabajos donde se requiere manejo de un gran volumen de datos y velocidad de procesamiento y los segundos cada vez más destinados a aplicaciones específicas con interacción con el exterior en tiempo real.

En la década de los 90 se introdujeron las memorias EEPROM y FLASH, que rápidamente fueron incluidas en los microcontroladores.

En la década de los 2000 y los últimos años, la competencia en el sector y el avance tecnológico ha permitido que el precio de los microcontroladores haya disminuido notablemente, pudiendo encontrarse algunos hoy en día incluso por menos de un dólar.

1.2.2 Características

1.2.2.1 Definición

Un microcontrolador es un circuito integrado que contiene una CPU, memoria y periféricos de entrada y salida, todo esto comunicado con buses. El encapsulado que contiene dichos elementos cuenta con pines para comunicar los periféricos con el exterior.

Como se ha mencionado anteriormente, en contraste con la gran potencia de cálculo y la alta velocidad de ejecución de un microprocesador, el microcontrolador está enfocado a realizar un número bajo de tareas específicas y a comunicarse en tiempo real con el entorno. Son predominantes en tareas donde hay que tener muy en cuenta el coste, tamaño y consumo del dispositivo, de ahí que su mayor aplicación sea en el control y funcionamiento de sistemas empotrados, es decir, en prácticamente cualquier dispositivo electrónico, desde juguetes y aparatos domésticos hasta máquinas industriales. En cualquier casa hay decenas de microcontroladores diferentes.

1.2.2.2 Arquitectura

Hay dos arquitecturas posibles para un microcontrolador, la arquitectura “Von Neumann” y la “Harvard”.

- Arquitectura Von Neumann: es aquella en la que la CPU se comunica con una memoria que contiene tanto los datos como las instrucciones.

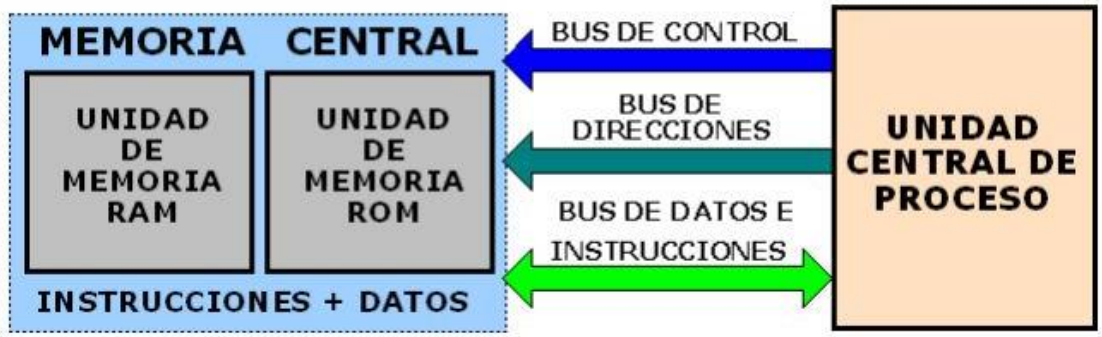

Figura 1-1. Arquitectura Von Neumann.

- Arquitectura Harvard: la CPU se comunica con la memoria de datos y la de instrucciones por separado y en paralelo.

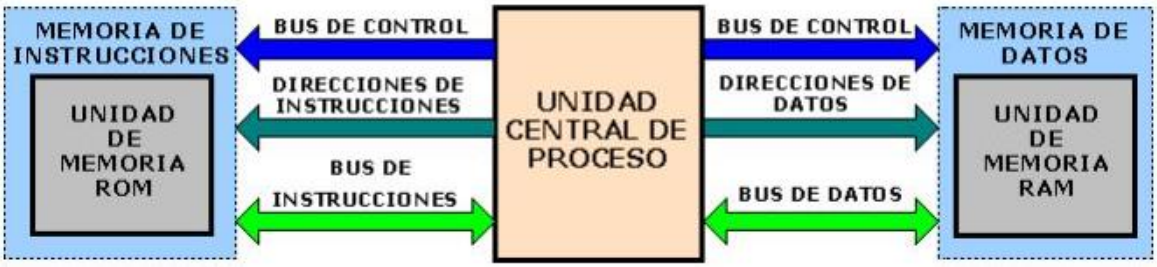

Figura 1-2. Arquitectura Harvard.

La principal diferencia en cuanto al funcionamiento es que la Von Neumann reduce el número de buses y por tanto también el coste de fabricación, mientras que la Harvard es significativamente más rápida, pues permite dimensionar los buses en función del tamaño de bits de cada memoria y acceder a ambas de manera simultánea.

Siendo la arquitectura Harvard la predominante en los microcontroladores, la estructura básica de uno es la siguiente:

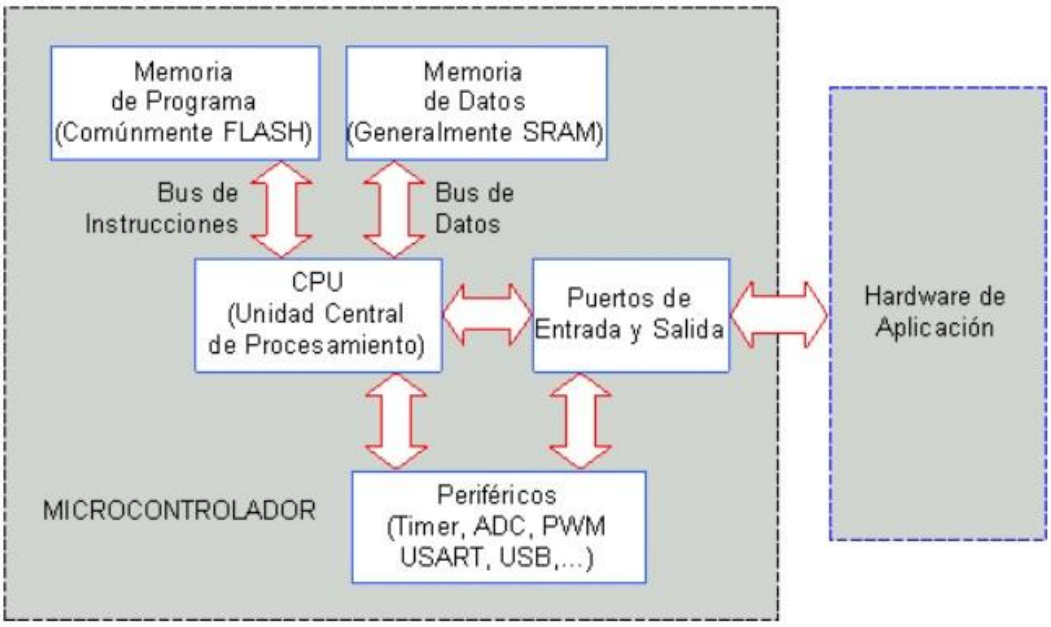

Figura 1-3. Estructura básica de un microcontrolador.

1.2.2.3 CPU

La Unidad Central de Procesos es el cerebro de cualquier ordenador. Es la parte del microcontrolador que se encarga de controlar y ejecutar las instrucciones.

En sus comienzos, la CPU era un solo elemento o núcleo, conformado a su vez de 3 componentes básicos (además de los buses internos): la Unidad de Control (UC), la Unidad Aritmético Lógica (ALU) y los registros. Con el tiempo se llegó a adoptar la tecnología multi-núcleo, que consiste en que la CPU tiene varios núcleos que pueden actuar de forma simultánea.

La ALU es la unidad encargada de realizar las operaciones básicas aritméticas y lógicas (sumas, restas, desplazamientos, comparaciones, etc).

La UC es la encargada de controlar el flujo de información a las distintas unidades cuando estas las necesite. Maneja el orden en que se ejecutan las instrucciones, enviando y recibiendo señales del resto de unidades (por ejemplo, decide qué operación debe realizar la ALU y cuándo) y controla el acceso a la memoria principal.

En cuanto a los registros, son pequeños espacios de memoria donde se almacenan los resultados de la ejecución de las instrucciones (además de datos que se cargan de las memorias externas), los cuales serán usados por el resto de los elementos de la unidad de procesamiento. A pesar de su pequeño tamaño, el registro influye en gran medida en el funcionamiento de la CPU de un microcontrolador, pues limita el tamaño de otros elementos. Por ejemplo, en uno con tamaño de registro de 16 bits se necesitará de un solo ciclo para ejecutar una suma de 16 bits, mientras que en uno con tamaño de 8 se necesitarán de varios ciclos, siendo por tanto más lento.

Por último, los buses son el medio por el que la CPU se comunica con el exterior. Son 3:

- Bus de direcciones: para seleccionar el dispositivo o espacio de memoria con el que se va a trabajar.
- Bus de datos: para el flujo de los datos que se van a utilizar en las instrucciones.
- Bus de control: para controlar las instrucciones que se realizan en las distintas unidades.

En la siguiente figura se ilustra la estructura básica de una CPU que se comunica solo con una memoria (arquitectura Von Neumann):

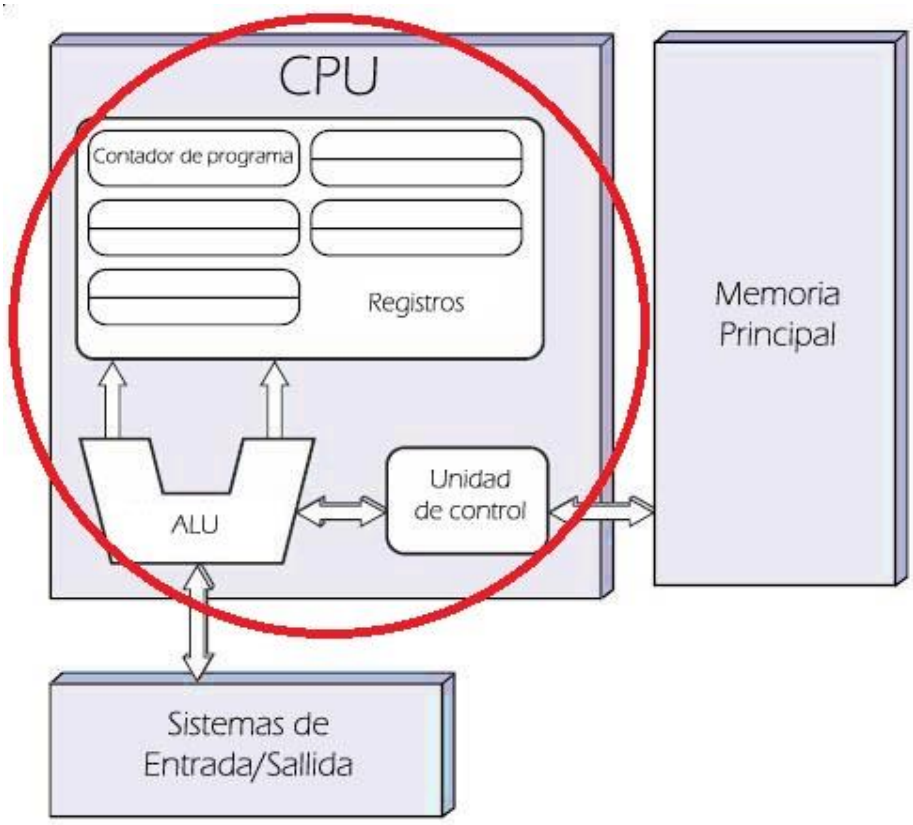

Figura 1-4. Estructura básica de comunicación de CPU con memoria y sistemas de E/S.

1.2.2.4 Memoria

El tipo de memoria más utilizada para los datos (memoria activa) es la SRAM, debido a que no necesita refrescarse como es el caso de la DRAM y aunque es más cara, no supone mucha diferencia porque no se usa una gran cantidad de esta memoria en los microcontroladores.

En cuanto a la memoria de instrucciones (memoria pasiva), la más utilizada es la FLASH, memoria tipo ROM que permite ser reescrita un gran número de veces y es más fácil de reescribir que sus antecesoras (EPROM, EEPROM, etc).

1.2.2.5 Periféricos

Los periféricos son los dispositivos o elementos de hardware con los que el microcontrolador se comunica con el exterior. De la gran variedad que hay, los más comunes en la arquitectura interna de un microcontrolador los siguientes:

- De entrada y/o salida (E/S): estos periféricos son los encargados de comunicar el microcontrolador con el exterior en cuanto a propósitos de carácter general, principalmente leer y escribir datos. Suelen estar agrupados en puertos de 8 bits de longitud.
- Temporizadores/contadores: encargados de llevar los periodos de tiempo del oscilador interno (temporizadores) o del conteo de eventos externos, ya sea de un reloj externo o de otro tipo de eventos (contadores).
- Perro guardián ("Watchdog"): se encarga de reiniciar el sistema si este se bloquea. Esto se consigue porque el watchdog es un contador con un valor alto inicial que va decrementándose pero que se refresca constantemente, a no ser que se bloque el sistema, momento en el cual dejará de decrementarse y cuando llegue a 0 mandará una señal de reset.
- Protección ante fallo de alimentación ("brownout"): sirve como otro mecanismo de protección, esta vez haciendo reset cuando la tensión de alimentación del sistema caiga por debajo del umbral mínimo.
- Modo de bajo consumo: tiene como objetivo "dormir" (detener el reloj) al microcontrolador o a ciertas partes de él mientras esté esperando alguna señal externa (interrupción), disminuyendo el consumo.
- Controladores de interrupciones: administran la ejecución de tareas por interrupciones.
- Convertidor analógico/digital (CAD) y digital/analógico (CDA): como su propio nombre indica, transforman un tipo de señal en otra.
- Comparador: Son circuitos analógicos basados en amplificadores operacionales. Comparan dos señales analógicas y dan como salida un '0' o '1' lógicos en función de dicha comparación. Especialmente útiles para detectar cambios en señales de entrada.
- Modulador de ancho de pulso ("PWM"): nos permiten modificar el ancho de pulso de una señal periódica. Es decir, establece cuanto tiempo la señal es positiva dentro del periodo que hay entre un pulso y el siguiente. Este periférico es de especial importancia en este trabajo, pues es el que vamos a usar para determinar la potencia que transmitimos a los ventiladores: a mayor temperatura, más potencia debemos dar al ventilador, luego mayor será el ancho de pulso de este.
- Puertos de comunicación: permiten que el microcontrolador se comunique con otros dispositivos externos. En función del protocolo de comunicación que se utilice, se distinguen los siguientes tipos:
    - Puerto serie: transmite información bit a bit. Está dedicado principalmente a la comunicación con un pc o con otro microcontrolador. Puede ser UART (comunicación asíncrona) o USART (si también permite comunicación síncrona).
    - SPI ("Serial Peripheral Interface"): es un protocolo de comunicación basado en la comunicación "master-slave" (maestro-esclavo). El maestro, por ejemplo, el propio microcontrolador, es el que ejecuta las órdenes que quiere que cumpla el esclavo, por ejemplo, un sensor de temperatura u otro microcontrolador. Es un sistema eficaz cuando el esclavo se encarga de tareas específicas y sencillas. El funcionamiento se basa en las líneas Tx y Rx: siempre que el reloj del maestro genera un pulso, la línea Tx transmite un mensaje y la línea Rx recibe otro al mismo tiempo. Por ejemplo, el maestro envía una orden por Tx y el esclavo

envía un dato al maestro por Rx. Esto sin embargo supone el uso de "dummy data" (datos que no sirven para nada) en ciclos de reloj en los que no se está haciendo nada.

- CANbus: es un protocolo "multi-drop", es decir, que puede transmitir información a diversos nodos en lugar de a uno solo, lo cual supone una clara mejora. Al ser el periférico clave en este proyecto, hablaremos más extensamente de él más adelante.
- Otros: USB, I2C, Ethernet, etc.

1.3 CANbus

1.3.1 Historia

Fue desarrollado por la empresa alemana BOSCH a partir del año 1983 y fue finalmente lanzado al mercado en el año 1986. Fue originalmente concebido para el uso dentro de la electrónica de los vehículos de pasajeros, aunque hoy en día se utilizan en diversas industrias, desde dispositivos médicos hasta dispositivos domésticos como ascensores o sistemas de ventilación.

1.3.2 Funcionamiento del bus

CANbus es un sistema de comunicación por bus "multi-master" y "multi-drop" enfocado a la comunicación en sistemas distribuidos con varios nodos. Esto es, todos los nodos pueden recibir mensajes de cualquier otro nodo. Por tanto, hay que aclarar que, aunque en este trabajo se hable de master y nodos, esto es solo por la función que desempeña cada placa y no porque haya esa jerarquía. La información que se transmite se descompone en paquetes de mensajes estandarizados. Los mensajes, que se transmiten en tramas de bits, tienen un número de identificación único dentro del sistema que permite saber qué nodo envía el mensaje y qué nodo o nodos lo reciben. Esto permite una comunicación más sencilla entre dispositivos de distintos fabricantes.

El bus CAN trabaja negado respecto a la entrada del nodo emisor y la salida del receptor. Es un bus diferencial: el bus CAN consta de dos señales, CAN H y CAN L. Cuando el emisor transmite un valor lógico '1', ambas señales están en estado "recesivo" de normalmente 2.5 V. Cuando el bit que se transmite es un '0', las señales muestran un estado "dominante", con CAN H a 3.5 V y CAN L a 1.5 V. Este sistema diferencial es balanceado, de modo que la corriente fluye de igual magnitud, pero en sentido contrario en ambas líneas, produciéndose una cancelación de campo electromagnético que reduce considerablemente el ruido. La siguiente figura se muestra lo anteriormente explicado:

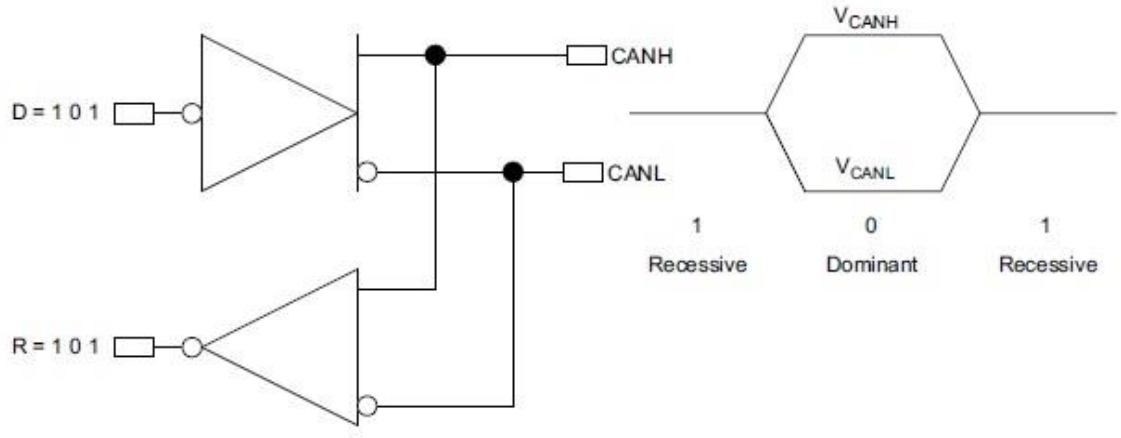

Figura 1-5. Funcionamiento de CANH y CANL.

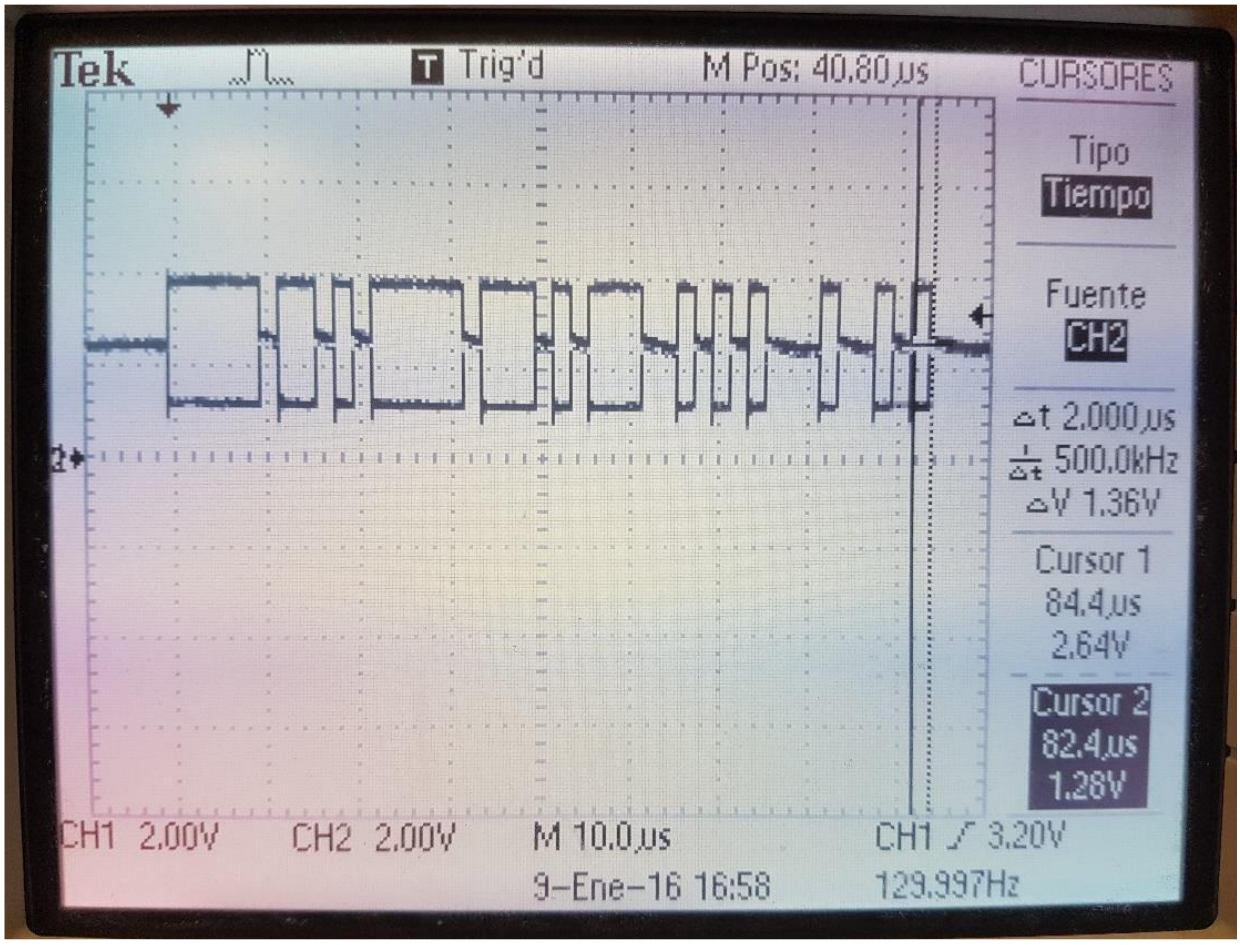

Figura 1-6. Ejemplo de trama de mensajes en CANbus.

En cuanto al apartado físico, cada nodo cuenta con un controlador CAN que se comunica con el transceptor físico, que se comunica a su vez directamente con el bus:

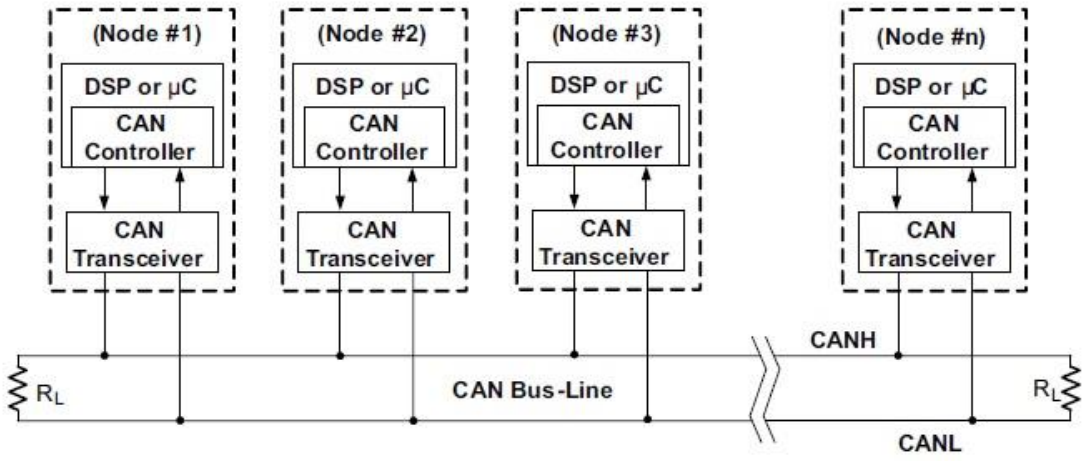

Figura 1-7. Comunicación entre nodos por medio del bus CAN.

1.3.3 Características

De lo mencionado anteriormente acerca del tipo de comunicación que conlleva, se derivan una serie de características fundamentales para el entendimiento del sistema:

- Prioridad en los mensajes: este es un aspecto muy característico en este tipo de comunicación. La información es transmitida por cualquier nodo de la red. Esto puede producir que al mismo tiempo varios nodos quieran transmitir un mensaje. La confrontación es solucionada con una parte de la trama de bits que componen el mensaje dedicada a dar prioridad mediante un número de identificación. A menor valor de este conjunto de bits (más ceros), más tiempo se mantiene al bus en nivel dominante. En el instante en que se produce la "colisión" entre los dos nodos que estaban transmitiendo a la vez, esto es, uno sigue dominante y otro recesivo, el nodo que ha pasado a recesivo detecta esta colisión y deja de transmitir, "ganando" el otro nodo. En la siguiente figura se muestra cómo un nodo obtiene la prioridad frente a otro:

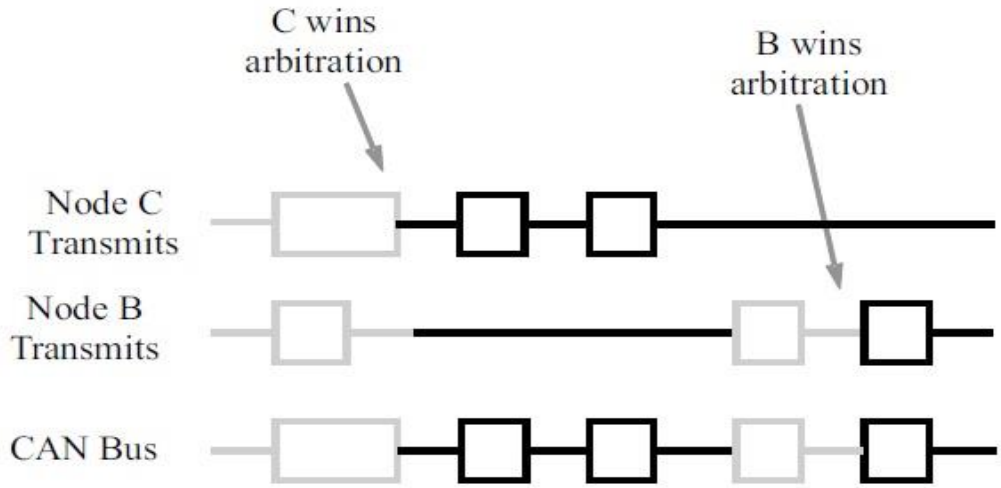

Figura 1-8. Prioridad de mensajes en CAN.

- Red multiplexada: reduce considerablemente el cableado.
- Robustez: se trata de un sistema muy resistente al ruido, en gran parte por la contraposición de las señales H y L y por la posición entrelazada de los cables. También es un sistema muy resistente a los errores, con mecanismos de detección de fallos y desacoplamiento de nodos consistentes en errores.

El resto de características y especificaciones no se incluyen aquí al no ser de gran importancia para el desarrollo y entendimiento del trabajo.

1.4 Programas utilizados

1.4.1 Code Composer Studio

Code Composer Studio (abreviado CCS) es un entorno de desarrollo de Texas Instruments dedicado a la programación y creación de proyectos con microcontroladores y microprocesadores de su catálogo. Se puede programar en lenguaje C y C++. Por tanto, es el que se ha utilizado en este trabajo, concretamente la versión 9.1.0. Más adelante, en el apartado de software, se entrará en más detalle respecto al programa.

1.4.2 KiCad

KiCad es un software de código abierto dedicado al diseño de circuitos impresos (PCB), similar al Eagle. Cuenta con un apartado dedicado a los esquemas, donde se dispondrán las conexiones necesarias entre los distintos elementos sin importar la distribución espacial de los mismos; y con otro dedicado al diseño del circuito impreso, donde sí distribuiremos cada componente en un lugar específico. Es el programa elegido para el diseño de la placa sobre la cual se montan el transceptor MCP2562, los sensores y los ventiladores, además del resto de componentes necesarios para el funcionamiento de estos.

2 HARDWARE

En este apartado se van a tratar los aspectos fundamentales de los dispositivos y sus componentes, además de mostrar imágenes de estos y de cómo se conectan para llevar a cabo los objetivos propuestos.

2.1 LaunchPAd Tiva™ C Series TM4C123G

La "Tiva™ C Series TM4C123G LaunchPad Evaluation Board" es la placa electrónica elegida para este trabajo. Cuenta con un gran número de elementos para poder hacer uso de todas las funcionalidades que ofrece, de los cuales para el proyecto son de interés los siguientes:

- Microcontrolador Tiva TM4C123GH6PM
- Control de movimiento PWM
- Conectores USB micro-A y micro-B
- LEDs RGB
- Dos interruptores (switch) de usuario
- Switch de reset
- Dos pares de columnas de 40 conectores de E/S en total, conectados a la mayoría de los pines GPIO del microcontrolador y que se usarán para conectar la placa al boosterpack que contiene el módulo CANbus.
- ICDI ("In Circuit Debug Interface"): permite hacer debug en programas como el CCS.
- Fuentes de alimentación seleccionables:
    - ICDI
    - Dispositivo USB

A continuación, se mostrarán la placa con indicaciones de algunos de los elementos mencionados y un diagrama de la misma:

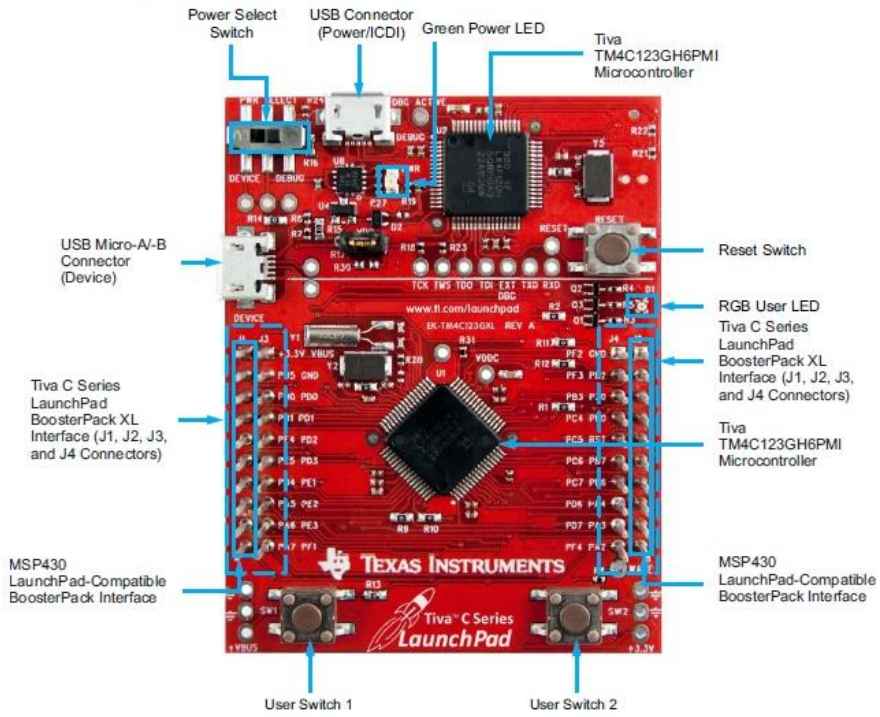

Figura 2-1. Tiva™ C Series TM4C123G LaunchPad Evaluation Board.

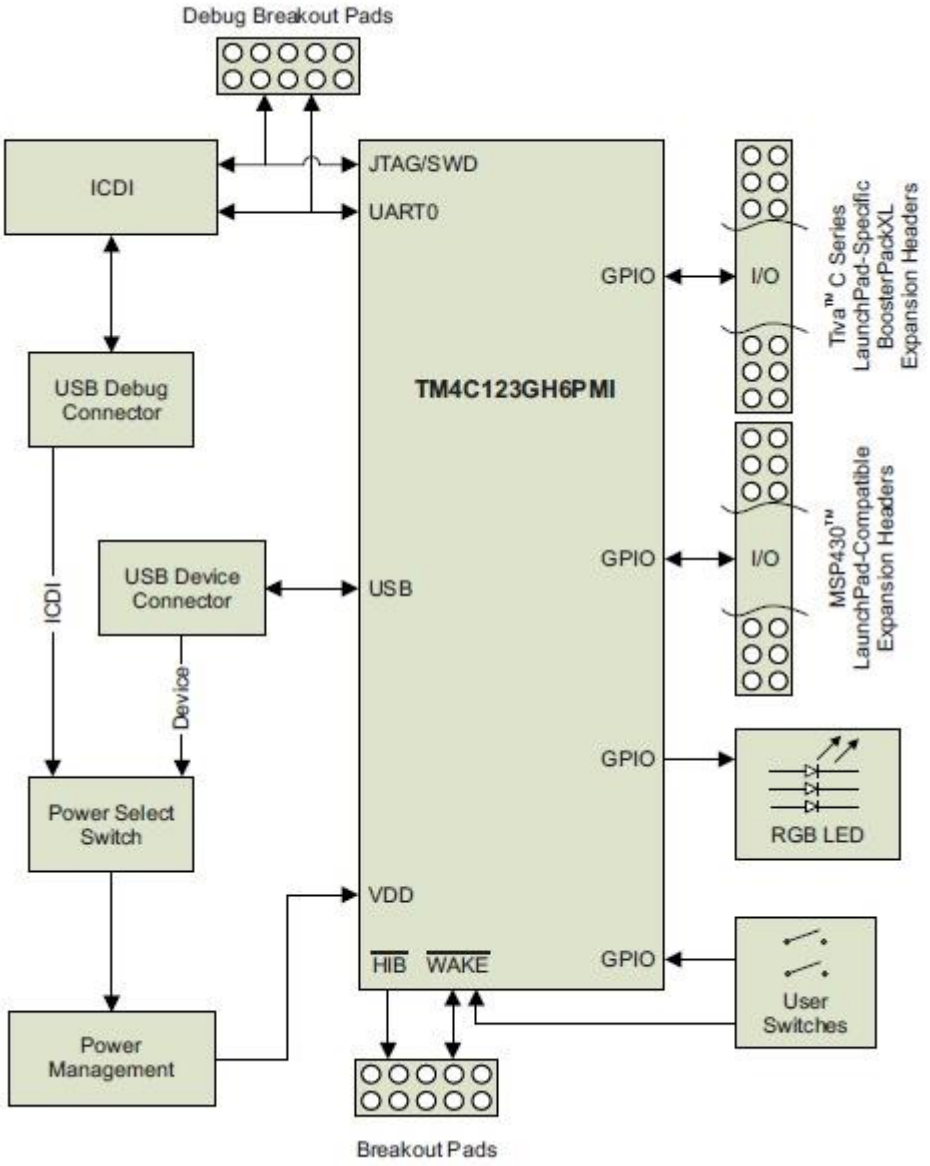

Figura 2-2. Diagrama de bloques de la tarjeta TM4C123G.

2.2 Transceptor MCP2562

El boosterpack (placa) sobre el que se instala el transceptor (transmisor y receptor) MCP2562 forma parte de las prácticas que se realizan en el M.I.E.R.A. Su objetivo es albergar dicho dispositivo junto con otros elementos necesarios para la realización de las prácticas, que son: pines para medir señales, pines de tierra y alimentación, trío de pines para conectar un servo, un potenciómetro, dos LEDs, tres condensadores, dos resistencias y un zócalo que se usa para poner encima de él el transceptor. Los pines están conectados por buses a las tiras de conectores que conectan esta placa con la Tiva C, que a su vez los conecta al microcontrolador de dicha placa.

En la siguiente figura se muestra una vista frontal y trasera de la placa:

Figura 2-3. Vista frontal del boosterpack con transceptor MCP2562.

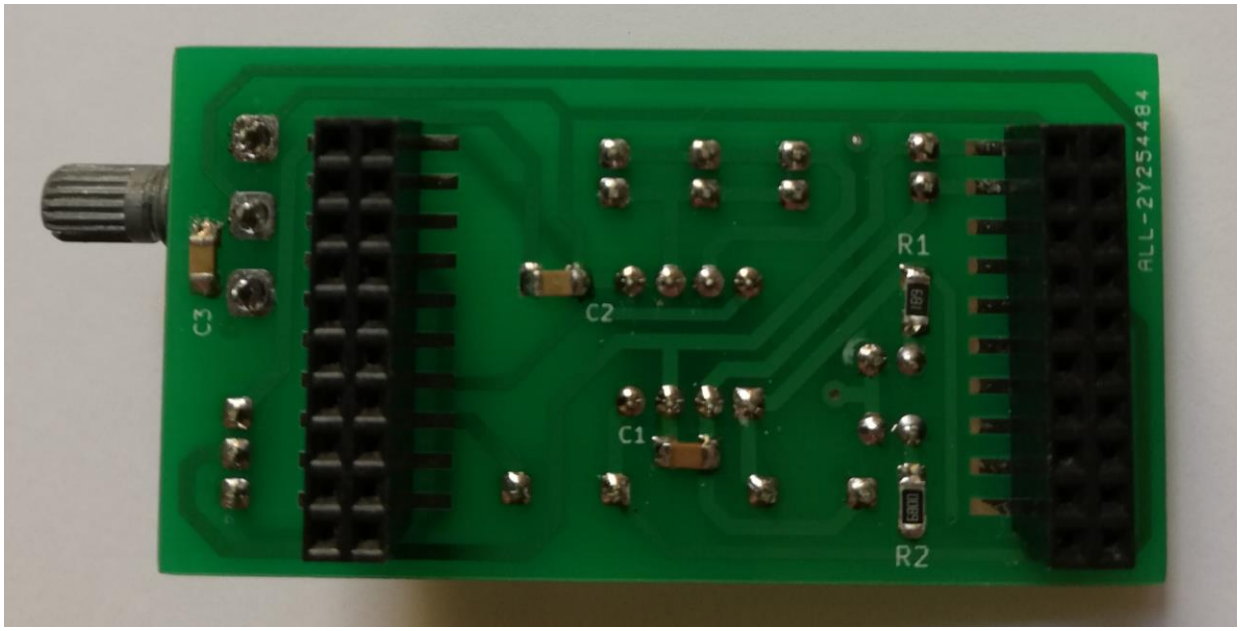

Figura 2-4. Vista trasera del boosterpack con transceptor MCP2562.

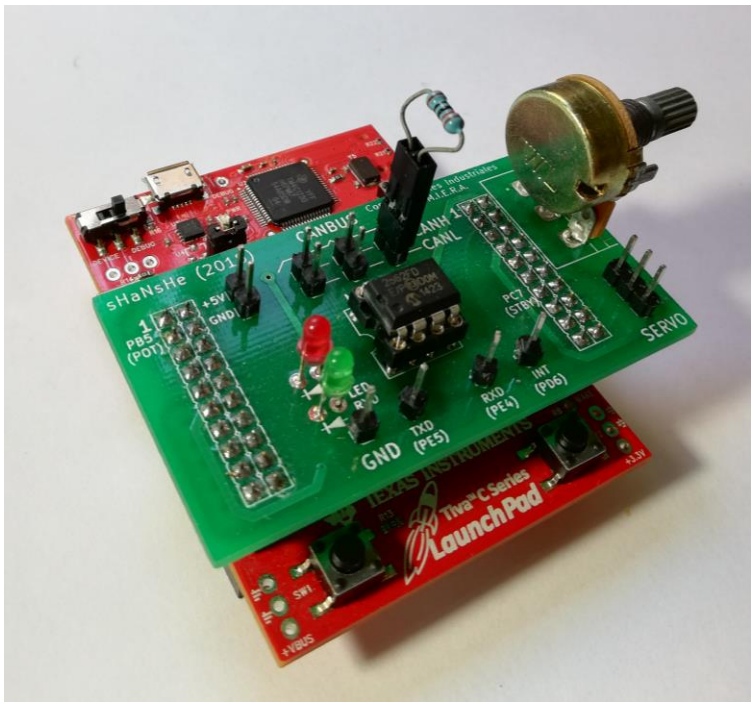

Figura 2-5. Boosterpack conectado a placa TM4C123G.

No es necesario profundizar más en cuanto a esta placa, aparte de apuntar que se utiliza como base para el desarrollo de otra placa más grande que se conecta en este trabajo a la tarjeta Tiva C para poder albergar los elementos que ya tenía y los necesarios para el proyecto.

En cuanto al transceptor CAN, sirve como interfaz entre el controlador del protocolo CAN y las dos líneas físicas del bus. Es un dispositivo de alta velocidad (operaciones a 1 Mb/s), con poco consumo (especialmente en el modo standby) y buena protección frente a faltas y ruido.

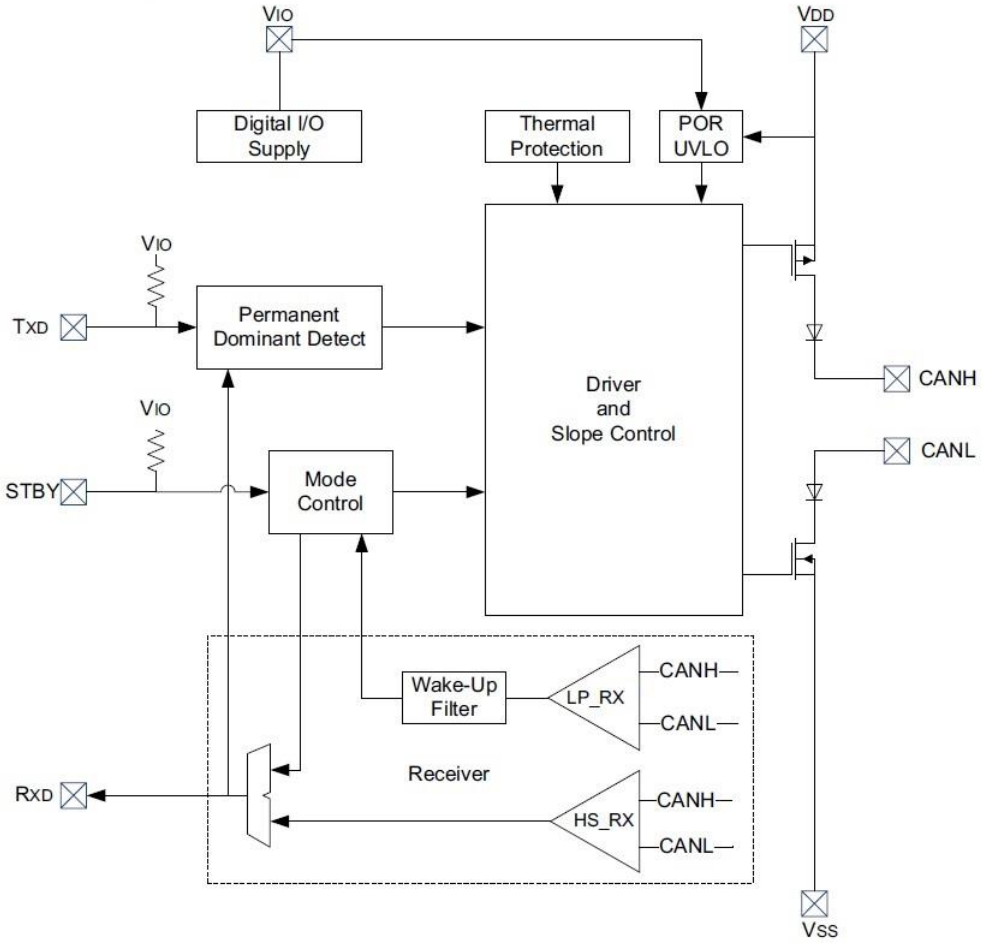

Figura 2-6. Estructura básica del transceptor MCP2562.

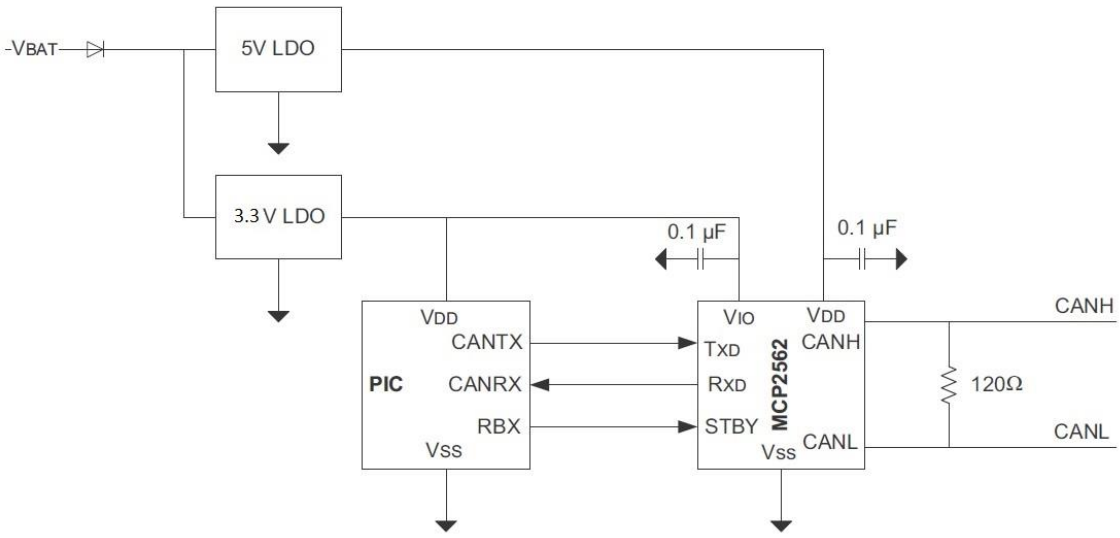

Figura 2-7. Conexión del transceptor MCP2562 con un microcontrolador.

El transceptor está conectado a una VIO de 3.3V para la comunicación con el microcontrolador y a 5V para CANH y CANL.

2.3 Boosterpack del trabajo

Como se ha dicho anteriormente, se parte del boosterpack utilizado en las prácticas para desarrollar otro que contenga los elementos necesarios para el desarrollo del trabajo. Los pasos para su fabricación están explicados en el apartado del software, por lo que en este apartado solo se van a mostrar imágenes de la placa y de cómo se conectan para llevar a cabo la tarea propuesta.

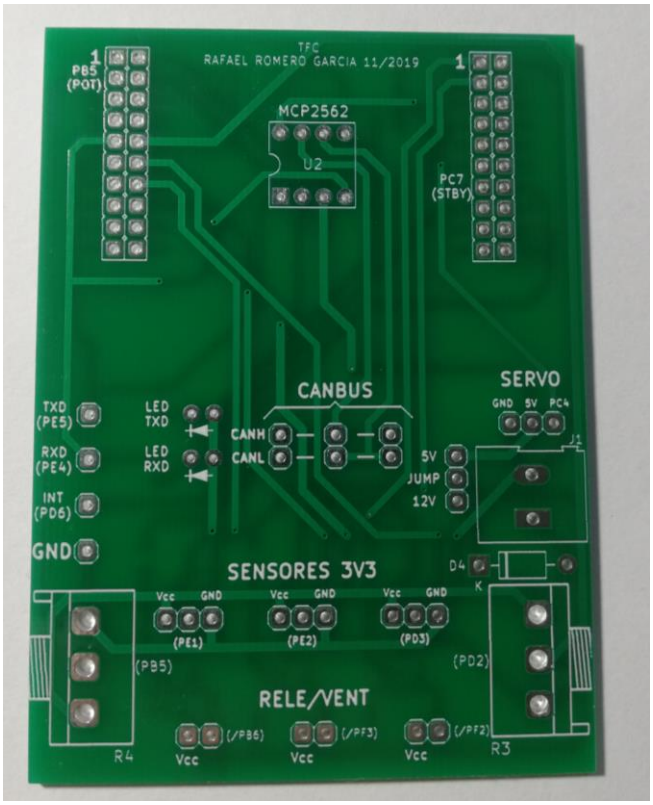

Figura 2-8. Vista frontal del boosterpack del trabajo sin componentes.

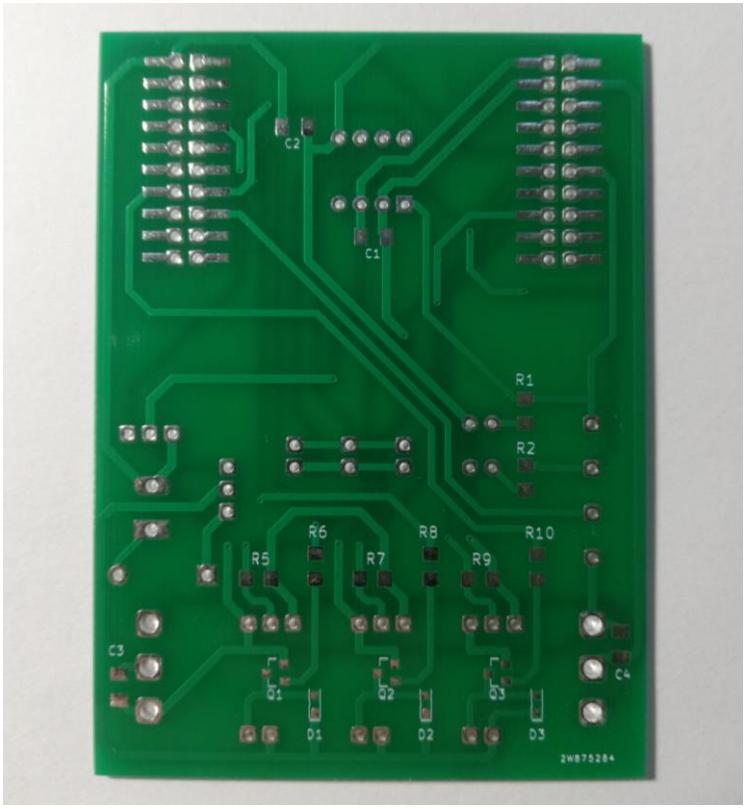

Figura 2-9. Vista trasera del boosterpack del trabajo sin componentes.

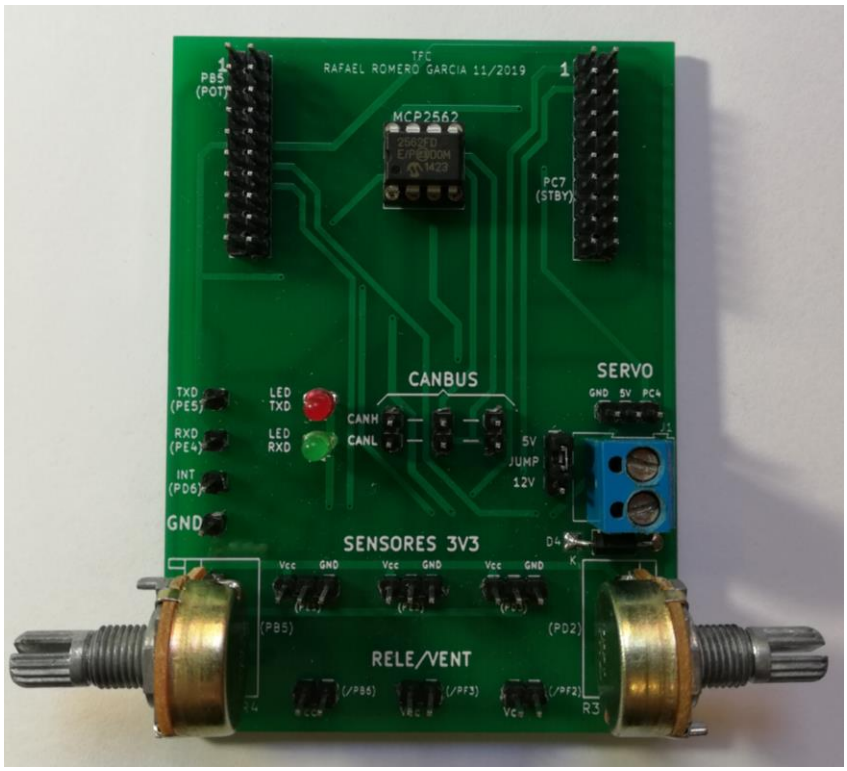

Figura 2-10. Vista delantera del boosterpack del trabajo con componentes.

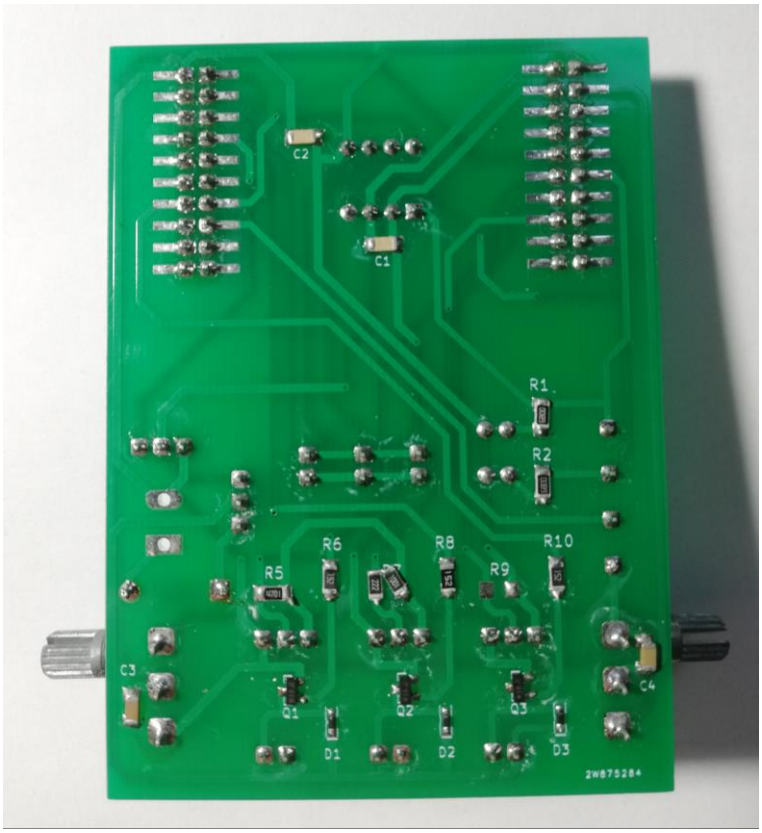

Figura 2-11. Vista trasera del boosterpack del trabajo con componentes.

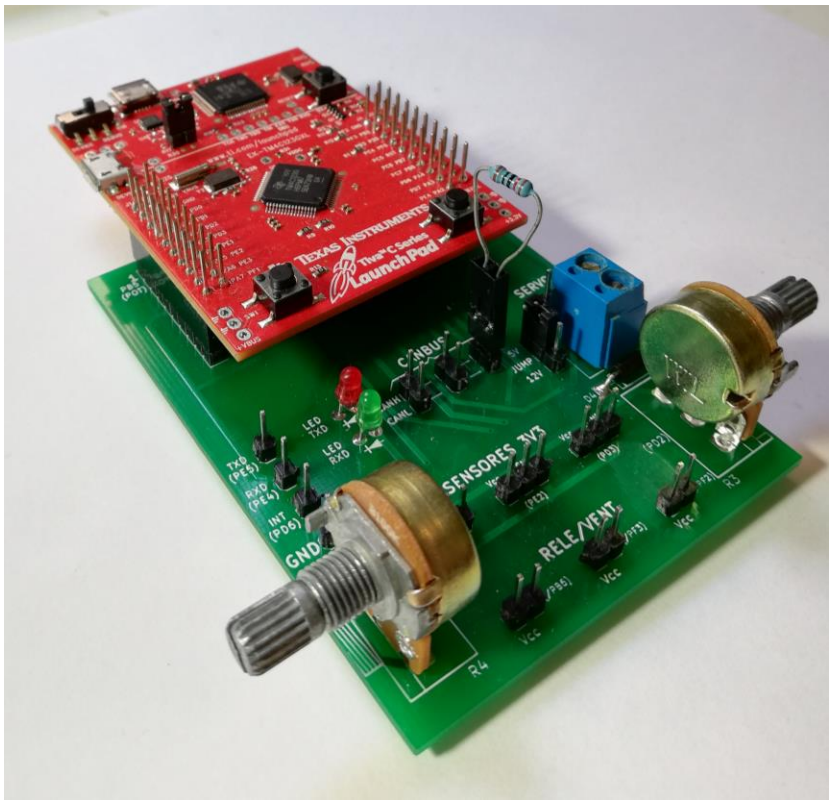

Figura 2-12. Boosterpack del trabajo conectado a la placa TM4C123G.

2.4 Conexionado de dispositivos

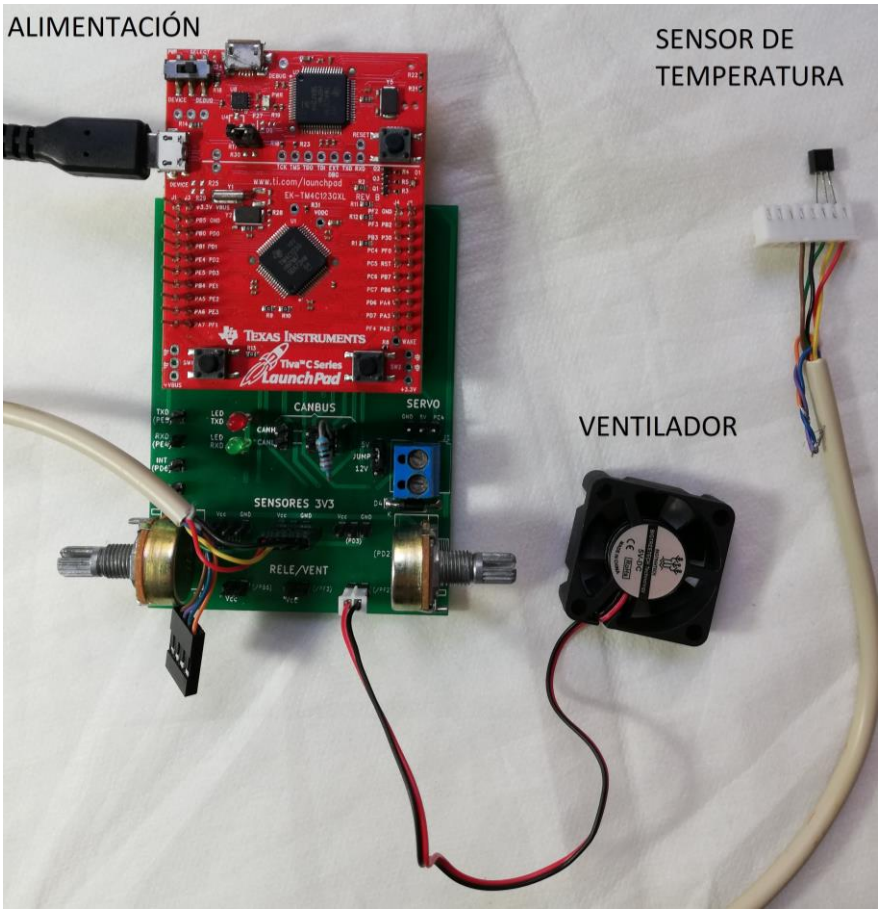

Figura 2-13. Conexionado del nodo con el ventilador y el sensor de temperatura.

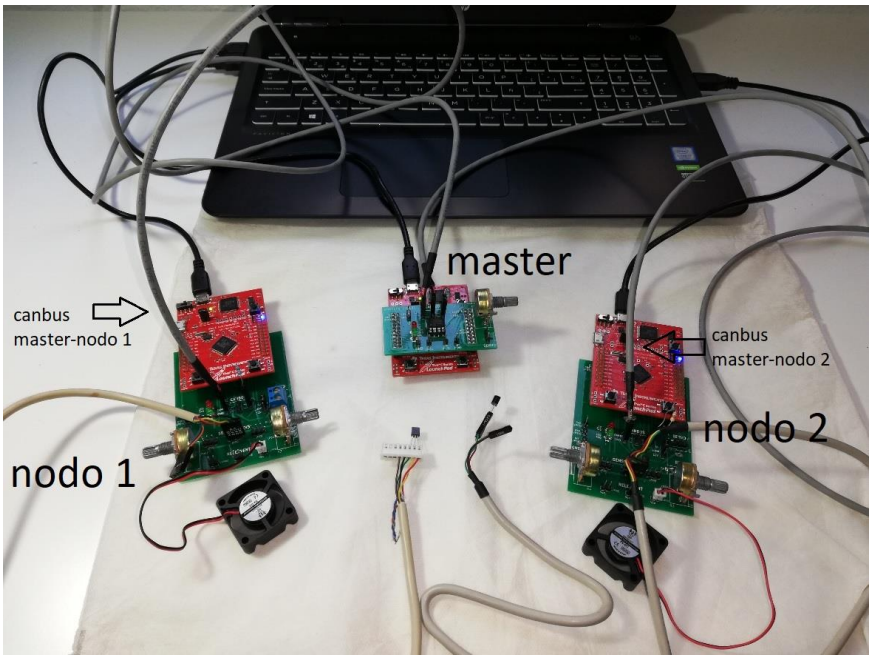

Figura 2-14. Conexionado completo master-nodos.

3 SOFTWARE

Este apartado está dedicado a todo lo relacionado con los programas que se utilizan y a los pasos que hay que seguir en cada uno de ellos.

3.1 Code Composer Studio

Como se dijo anteriormente, CCS es el programa utilizado para escribir y ejecutar el código que permite el funcionamiento de las placas de modo que se consigan los objetivos propuestos. Se tienen tres proyectos, uno para la placa que va a hacer de máster y dos para cada una de las placas que van a hacer de nodo, siendo estos dos últimos iguales con la única diferencia de que van a tener distintos números de identificación en el protocolo CANbus.

Tanto el proyecto para el máster como para el nodo van a ser prácticamente iguales en la parte de la comunicación entre ellos mediante el CAN, con la diferencia fundamental siendo que en el proyecto del nodo se incluye todo el código relacionado con el uso del sensor de temperatura y el ventilador.

CCS cuenta con todo lo necesario para la creación, compilación y "debugging" (depuración y ejecución) del código que es implementado en la placa, por lo que no es necesario descargar ningún programa adicional.

3.1.1 Primeros pasos

El primer paso es descargarse el software específico de la placa, el "TivaWare". En el App Center de CCS, se busca el software y descarga.

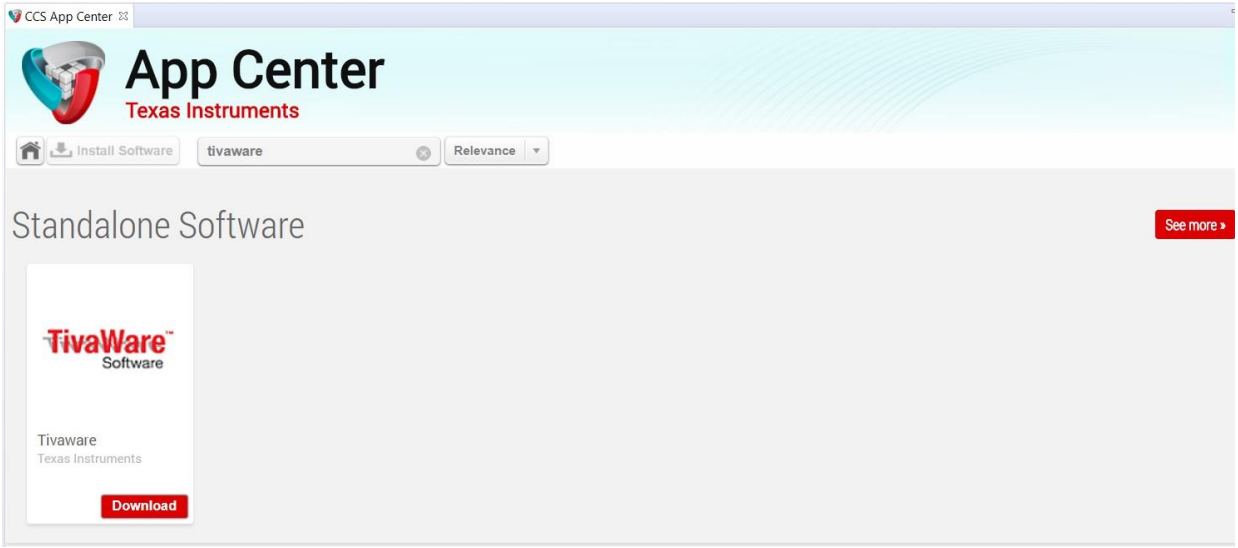

Figura 3-1. TivaWare.

Si es necesario, es conveniente actualizar el compilador ARM, también en el App Center.

Lo primero que se debe hacer en un proyecto en CCS es establecer las propiedades del mismo:

- En General:
  - Establecer el compilador, en este caso el TI v18.12.2.LTS, que viene por defecto en la versión 9 del CCS.
  - Establecer el dispositivo ("target"), en este caso, TM4C123GH6PM.
- En Build:
  - ARM Compiler>Include Options: incluir todos los directorios necesarios en función de los #includes que se van a utilizar en el código.
  - ARM Compiler>Predefined Symbols: añadir el símbolo PART_TM4C123GH6PM.
  - ARM Linker>File Search Path: incluir el directorio de la librería de los drivers de la placa.

El resto de propiedades se dejan por defecto.

3.1.2 Código

El código utilizado es de gran extensión y en su mayoría lo conforman archivos que contienen una gran cantidad de variables, estructuras, funciones, etc, necesarias para las distintas funcionalidades de las placas utilizadas. Por eso, en este documento solo se van a mostrar las partes más importantes relacionadas con el objetivo del proyecto. Estas son, las funciones que inicializan el sensor y hacen que se obtenga la temperatura, las que se usan para la comunicación con el CAN y el programa que las recoge para hacer la comunicación entre las placas.

El main.c tiene tres partes bien diferenciadas:

1. Includes, macros e inicialización de variables globales.
2. Definiciones de las funciones que se van a utilizar.
3. Programa principal, donde se configuran aspectos básicos del funcionamiento (frecuencia del reloj del sistema, master de interrupciones, etc) y se llama a las funciones previamente definidas, algunas antes del bucle infinito que va a simular el funcionamiento y otras dentro de este, además del manejo de las distintas variables necesarias, como la temperatura que devuelve el sensor o el pwm que el master establece para el ventilador.

Además del main.c, en la carpeta del proyecto del sensor también son de importancia los .c, .cpp y .h que se usan para el funcionamiento de este, de modo que en el main solo haya que llamar a las funciones necesarias que ya están definidas en estos archivos.

Para explicar el funcionamiento no se va a exponer el código sin más, si no que se va a hacer uso de diagramas de flujo y pseudocódigo de las partes más importantes, sin entrar demasiado en detalles técnicos. Aun así, se incluirá como ejemplo en un anexo el código del main del sensor, así como el código completo como material adicional.

3.1.2.1 CAN bus

El funcionamiento de la comunicación con CAN Bus se basa en 2 funciones usadas en el main.c, que además se sirven de otras funciones y variables que extraen de otros ficheros, principalmente del can.c. El módulo CAN que se usa en este trabajo es el CAN0, habiendo otro más que no usaremos (CAN1). Estas 2 funciones son:

- InitCAN0: se inicializan las variables y pines necesarios, así como ciertos aspectos del módulo, como la frecuencia a la que trabaja. También se inicializan el objeto de recepción y el de transmisión, que son los que van a portar la información que se transmite. El pseudocódigo de la función es el siguiente:

```
función InitCAN0
{
    Habilitar puerto E
    Configurar PE4 como pin de recepción del módulo CAN0
    Configurar PE5 como pin de transmisión del módulo CAN0
    Habilitar PE4 y PE5 como pines de tipo CAN
    Habilitar periférico CAN0
    Inicializar el controlador del CAN
    Configurar la frecuencia del CAN
    Habilitar las interrupciones del módulo CAN0
    Habilitar las interrupciones del CAN en el registro NVIC
    Habilitar el CAN
    Inicializar el objeto de recepción RXOBJECT
        ID
        máscara
        banderas
        tamaño
        CANMessageGet (función que forma el mensaje)
    Inicializar el objeto de transmisión TXOBJECT
        ID
        máscara
        banderas
        tamaño
        puntero que apunta a la variable donde almacenaremos el objeto
}
```

- CAN0IntHandler: es la función que maneja las interrupciones que se producen debido al CAN. Básicamente controla las banderas que se van a usar en el main para manejar la comunicación entre los dispositivos:

```
función CAN0IntHandler
{
    Declarar variable status
    Almacenar en status el estado del CAN y la causa de este

    Si (status=interrupción)
        Almacenar en una variable el error para gestionarlo
        en la función CAN0ErrorHandler
    Si (status=id del objeto de recepción)
    {
        Bajar bandera del CAN
        Subir la que se usa en el bucle infinito para denotar llegada
        de objeto de recepción
```

```
        Poner errores a 0
    }
    Si (status=id del objeto de transmisión)
    {
        Bajar bandera del CAN
        Subir la que se usa en el bucle infinito para denotar llegada
        de objeto de transmisión
        Poner errores a 0
    }
}
```

3.1.2.2 Sensor

La parte fundamental del código del sensor son las dos funciones que se usan en el main de cada placa, "setup" y "loop", que se van a encargar de inicializar el sensor y de almacenar la temperatura que este lee, respectivamente. Cabe destacar que algunas de las funciones utilizadas para el funcionamiento del sensor están definidas dentro de una clase llamada "DS18B20", que será más tarde declarada como "ds".

Setup es una función que solo se encarga de llamar a otras dos funciones: "ds.InitGPIO", la cual simplemente inicializa el pin que corresponda al sensor como salida open-drain; y "findOW", que se encarga de encontrar el sensor dentro del 1-wire (dentro del 1-wire puede haber varios sensores). Dentro de findOW se implementa la comunicación del sensor (esclavo) con la placa que lo contiene (master). El proceso para empezar la comunicación se detalla en la función "reset", que devolverá un 0 si ha sido satisfactorio y es el siguiente:

```
función DS18B20::reset
{
    Establece el pin como salida open-drain ("habla" el master)
    Pone la línea a 0
    La mantiene durante 500 microsegundos con un delay, pues la
    mínima duración del pulso de reset del master es 480

    "Suelta" la línea (la pone a 1)
    Pone el pin como entrada (ahora "habla" el sensor)
    Delay de 56 microsegundos (el sensor debe esperar de 15 a 60)

    Si la lectura del pin devuelve un 1, return 1
    porque el sensor lo debería haber puesto a 0

    Delay de 250 microsegundos, pues el pulso de presencia del
    sensor puede durar de 60 a 240 (se asegura con 250)

    Si al volver a leer el pin se devuelve un 0, return error,
    porque debería valer 1 (ya habría soltado el sensor la línea)
```

```
        return 0
    }
```

Este procedimiento está detallado en el datasheet del sensor y hay que respetar los tiempos o de lo contrario será imposible la comunicación con el dispositivo. En la siguiente figura se ilustran los pulsos descritos anteriormente:

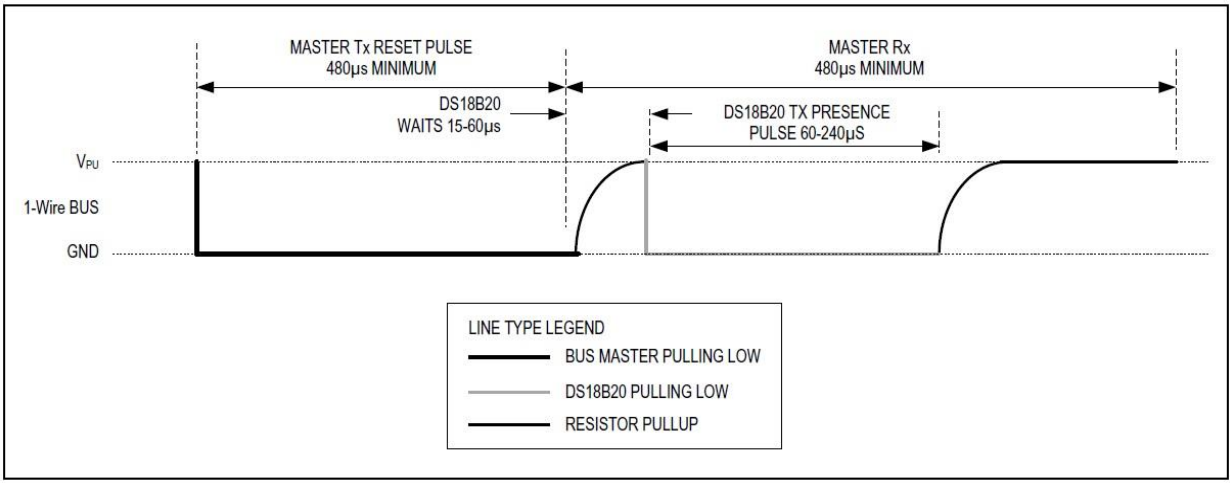

Figura 3-2. Intervalos de tiempo de inicialización de la comunicación con el sensor.

Como se puede apreciar, la comunicación se basa en el uso del pin asociado al sensor, "poniéndolo" a 0 o "soltándolo" a 1 durante distinto tiempo en función de lo que se quiera hacer.

Así, en el resto de operaciones, que el master escriba un 0 o un 1 al esclavo o los lea de este, es necesario que las funciones utilizadas en el código sigan, al igual que "reset", unos espacios de tiempo ("time slots") descritos también en el datasheet y mostrados en la siguiente figura:

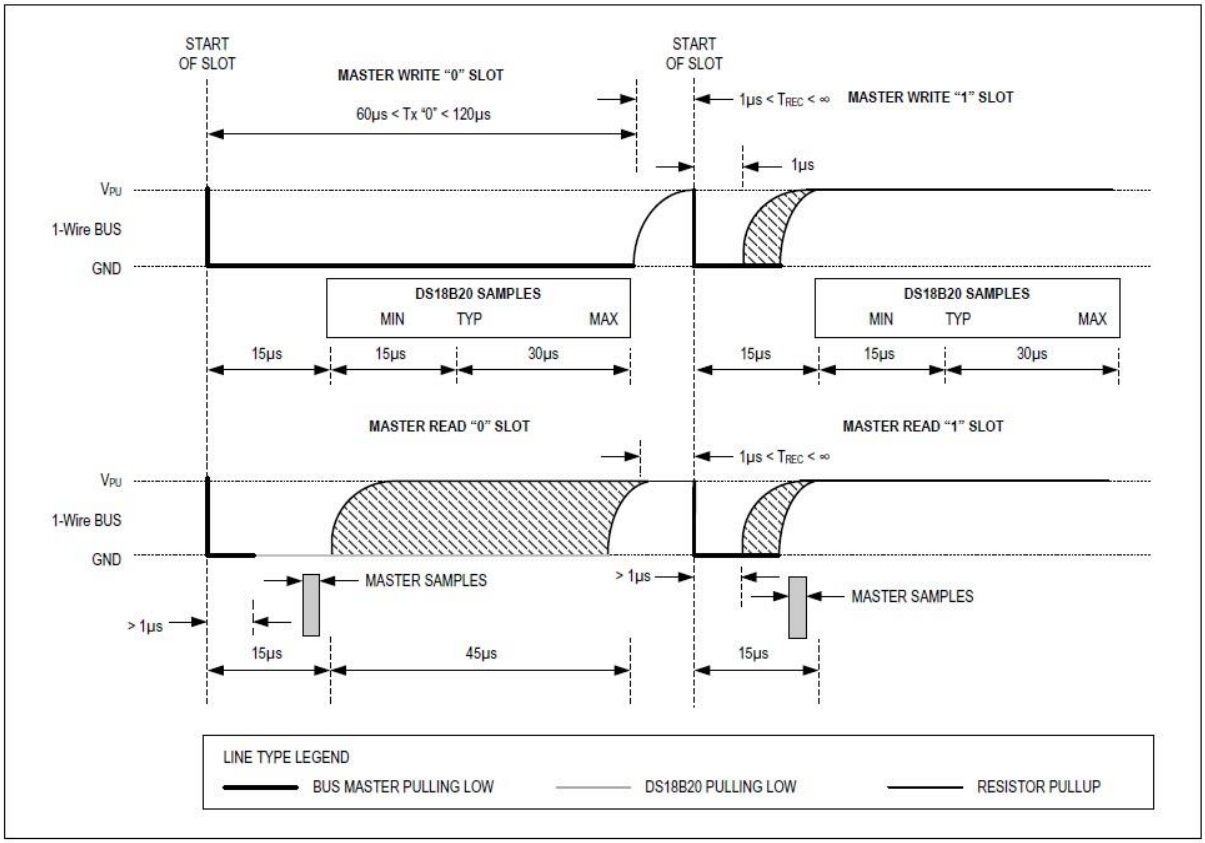

Figura 3-3. Intervalos de tiempo de escritura y lectura con el sensor.

Loop se encarga de llamar también a otras dos funciones, "readOW", que se encarga de leer el sensor; y "saveTemperature", que se encargue de guardar el valor de la temperatura. Además, en loop se ha implementado una máquina de estados para no interrumpir las interrupciones del timer, que se explicará más adelante.

3.1.2.3 Ventilador

El ventilador va conectado a un pin GPIO, que es inicializado en una función aparte, al igual que el CAN, utilizando uno de los módulos pwm que proporciona la placa:

```
función InitVent
{
    Se definen macros para el valor pwm máximo (255) y mínimo (1)
    Se habilita el puerto F
    Se habilita el módulo PWM1
    Configura el pin PF3 como PWM
    Configura el modo de funcionamiento del PWM1
        Modo (down)
        Periodo (PWMmax)
        pulso inicial (PWMmin)
    Se habilita el generador PWM correspondiente
}
```

3.1.2.4 Main

La función main (no todo el main.c), es decir, el programa principal, donde comienza a ejecutarse el código, tiene una estructura un poco menos lineal, por lo que se va a explicar mediante diagramas de flujo:

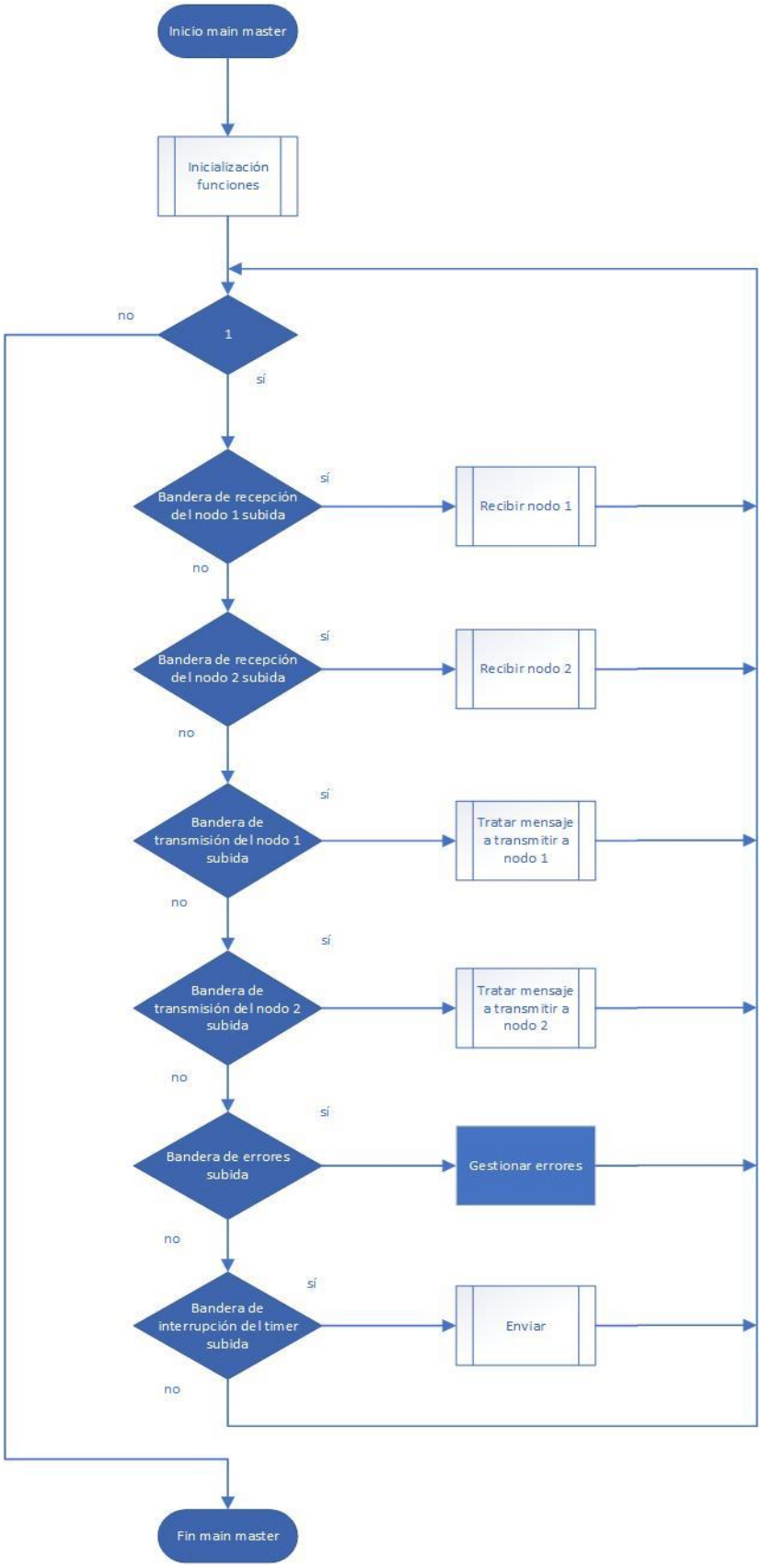

Figura 3-4. Diagrama de flujo del main del master.

Figura 3-5. Diagrama de flujo del proceso de recepción del master.

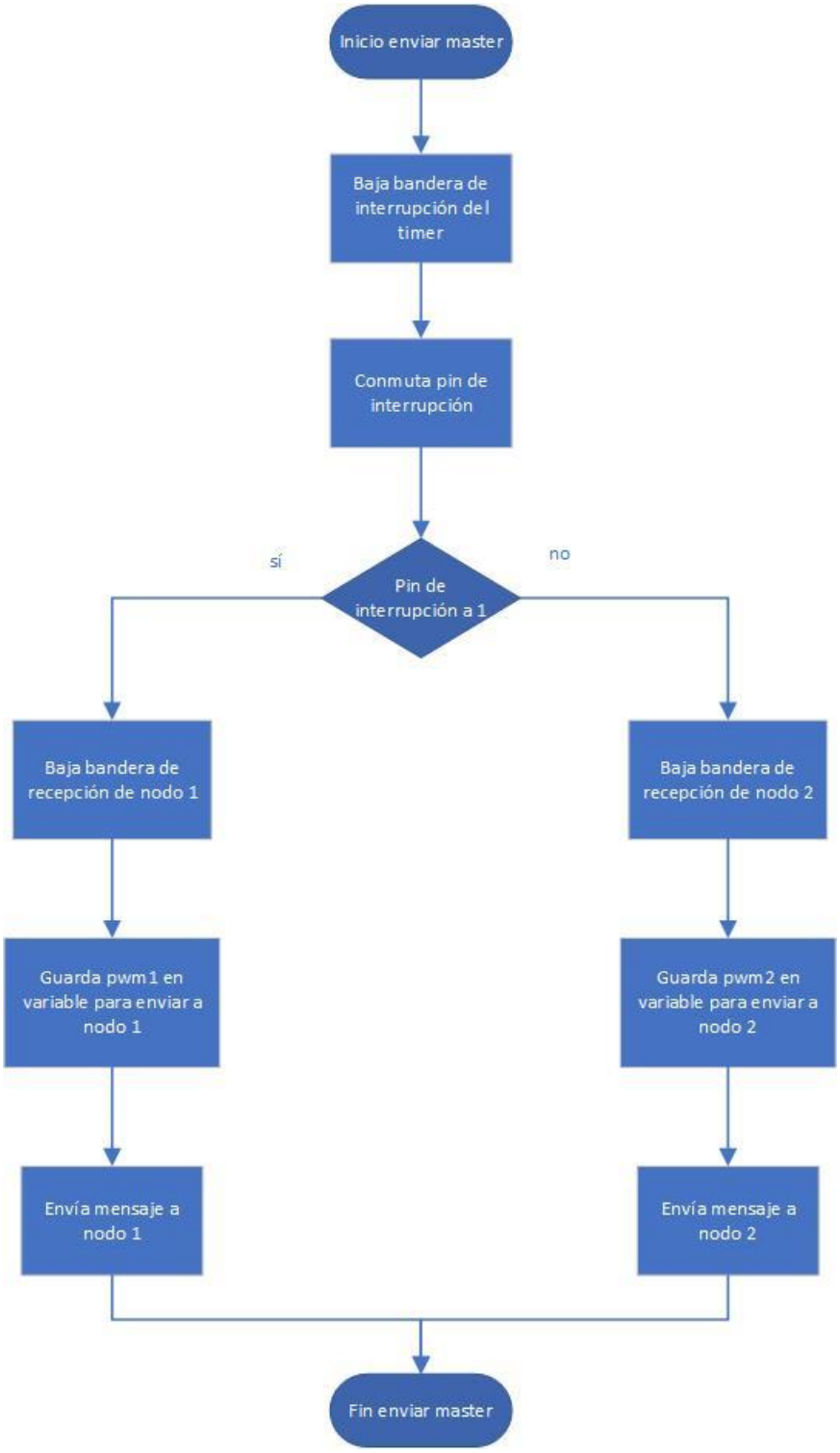

Figura 3-6. Diagrama de flujo del proceso de transmisión del master.

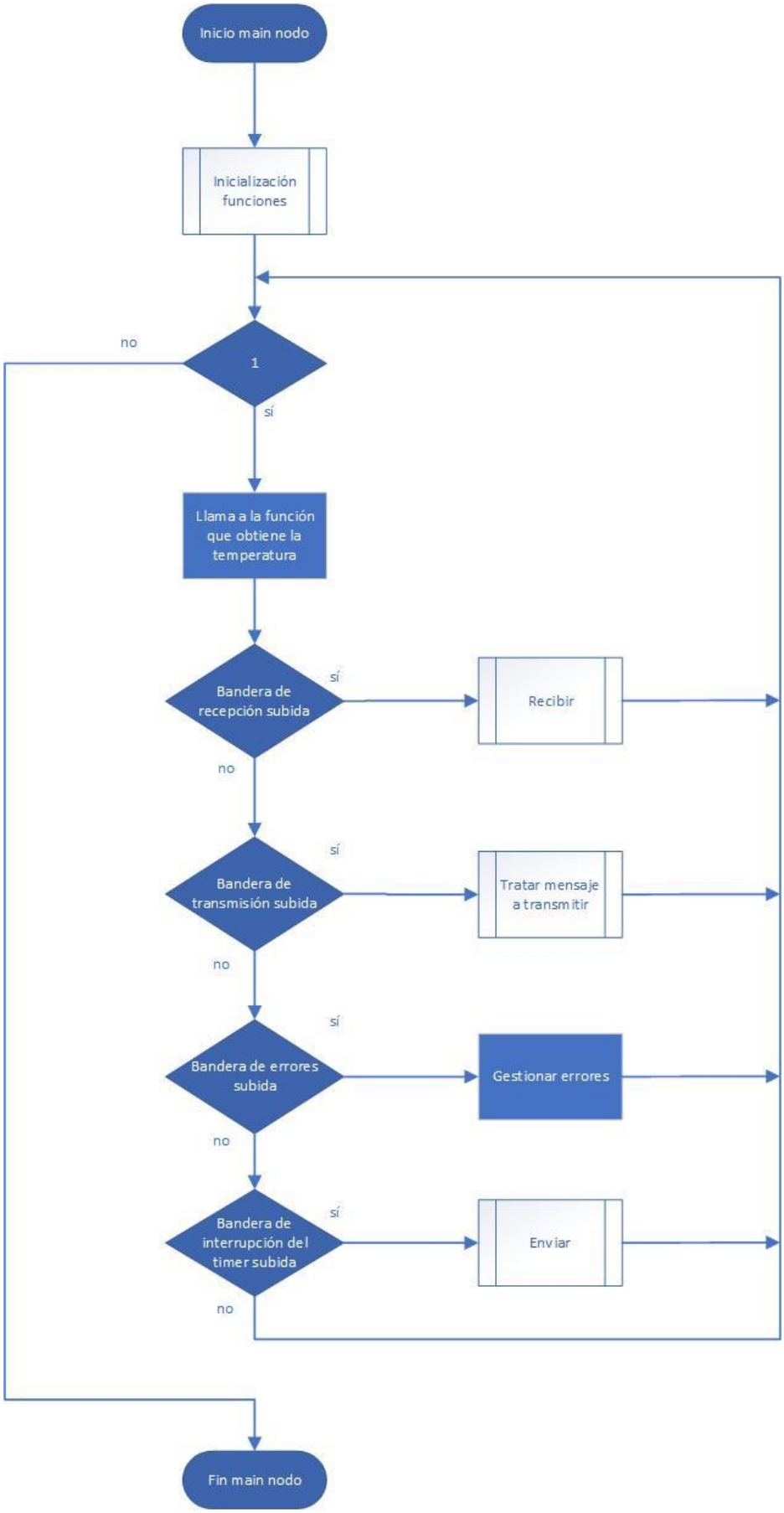

Figura 3-7. Diagrama de flujo del main de los nodos.

Figura 3-8. Diagrama de flujo del proceso de recepción de los nodos.

Figura 3-9. Diagrama de flujo del proceso de transmisión de los nodos.

3.2 KiCad

KiCad, como ya se ha mencionado antes, es el programa usado para la creación de la placa que contiene el transceptor CAN MCP2562, además de los pines para los sensores y ventiladores, manteniendo el resto de componentes necesarios para la realización de las prácticas del máster. Es decir, el trabajo que se ha hecho con este programa ha sido el de redistribuir los componentes que tenía el boosterpack que se utilizaba en las prácticas y adecuarlo al propósito del proyecto, intentando que el acabado sea lo más ordenado y compacto posible, teniendo en cuanta que parte de la placa va debajo de la tarjeta Tiva.

3.2.1 Esquema

El primer paso del trabajo con el KiCad es el esquema. Se seleccionan unas plantillas genéricas para cada elemento y se conectan según la configuración necesaria.

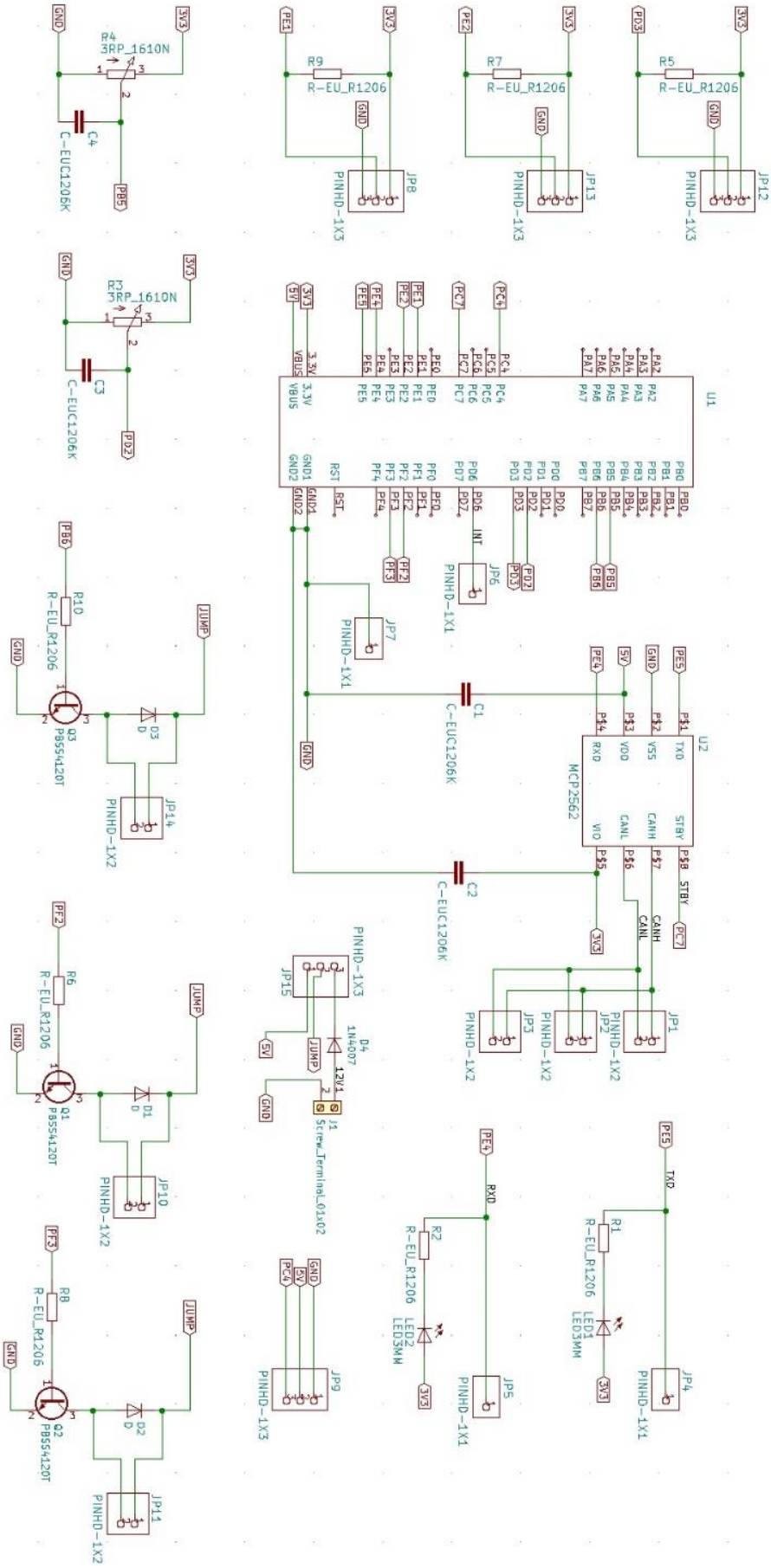

Figura 3-10. Esquema de KiCad: esquema completo.

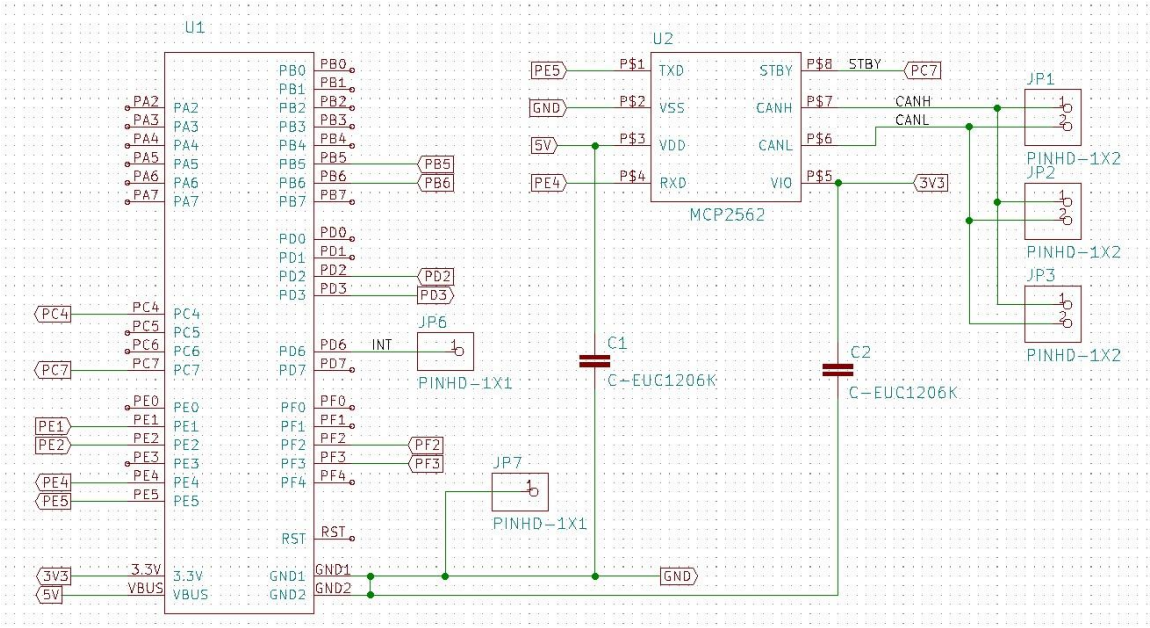

Figura 3-11. Esquema de KiCad: microcontrolador y el transceptor.

A la derecha de la figura 3-11 se muestran los 3 pares de conectores que representan el CAN-H y el CAN-L.

Los sensores vienen representados por 3 pines, uno conectado a Vcc (3.3V); otro conectado a una resistencia pull-up a Vcc y con salida al pin que se va a usar para controlar el transistor que va conectado al ventilador; y otro conectado a tierra. Esta configuración es la indicada por el fabricante del sensor para su funcionamiento y está mostrada en el datasheet del mismo.

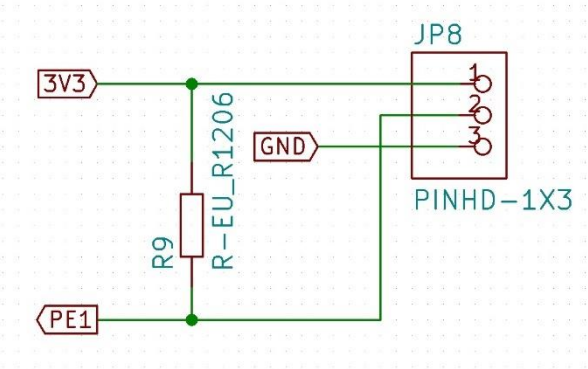

Figura 3-12. Esquema de KiCad: sensor de temperatura.

Los ventiladores van conectados a Vcc (5V o 12V en función del jumper) y al colector de un transistor BJT, el cual deja pasar corriente por el colector y el emisor si en la base hay suficiente. Como la base está conectada al pin configurado como pwm, cuando este valga 1 habrá paso de corriente por el ventilador (activándose) y cuando este valga 0 no habrá. Así, a mayor pwm, más tiempo estará a 1 la entrada por cada periodo, traduciéndose en más potencia en el ventilador.

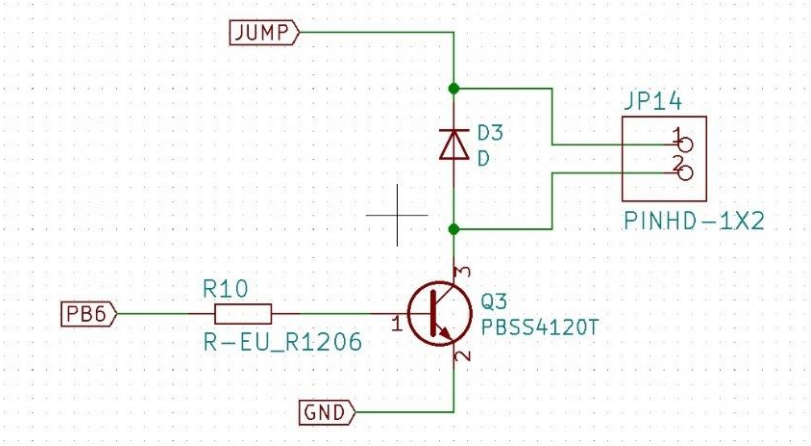

Figura 3-13. Esquema de KiCad: ventilador.

Nótese que se ha puesto el ventilador en paralelo con un diodo para evitar que si la intensidad fluye en sentido contrario no lo haga por el ventilador. También se pone una resistencia en la base de valor 1,5 kΩ, suficientes para que con el pin a 3.3V haya una corriente en la base que sature el transistor.

Para poder utilizar ventiladores tanto de 5V como de 12V, se ha añadido un "terminal block" (un bloque al que se pueden conectar los cables de una fuente externa) para poder conectar la placa a una fuente de 12V y cambiar entre 5 y 12 usando un jumper.

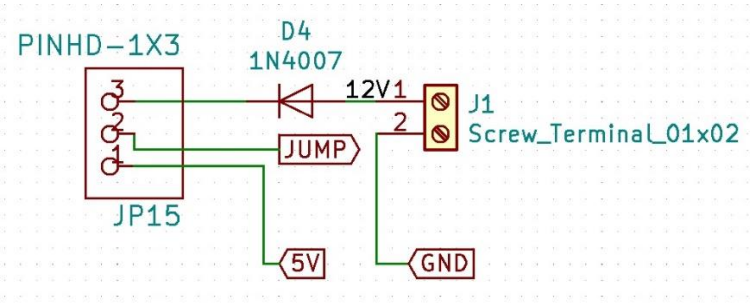

Figura 3-14. Esquema de KiCad: terminal block.

Nótese de nuevo el uso de un diodo para evitar que la corriente fluya en sentido contrario si se conectase la fuente al revés.

El resto de componentes representados en el esquema son los LEDs que se usan para ver cuándo hay transmisión y recepción en el CAN, el servo y los potenciómetros (material usado en las prácticas).

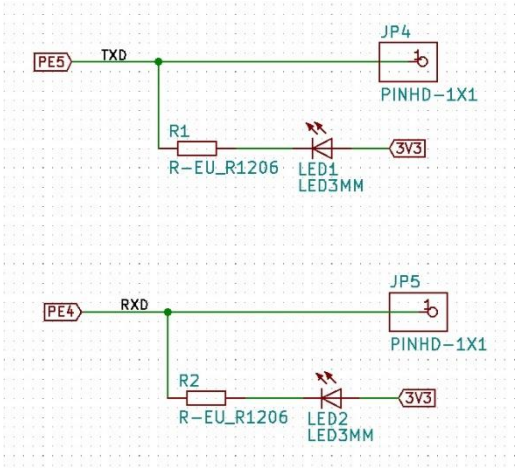

Figura 3-15. Esquema de KiCad: LEDs de transmisión y recepción.

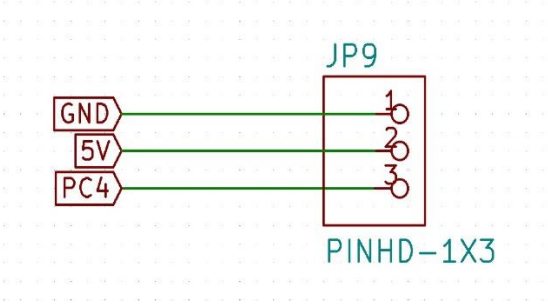

Figura 3-16. Esquema de KiCad: servo.

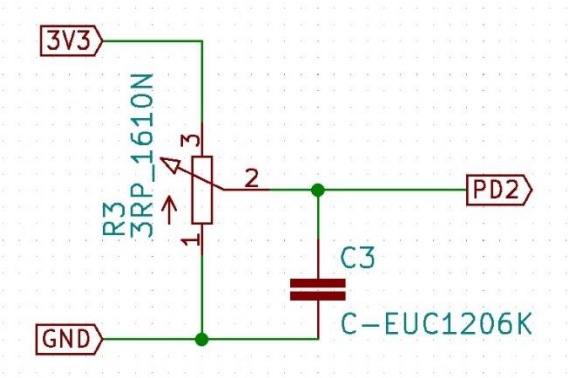

Figura 3-17. Esquema de KiCad: potenciómetro.

3.2.2 Huellas

Una vez completado el esquema se seleccionan las huellas de cada componente, que son el grabado que hace cada uno en la placa y sobre el cual se suelda cada uno. Cada huella pertenece a una librería y cada componente tiene una huella.

En el proyecto se han usado, aparte de las librerías por defecto del programa, dos librerías adicionales específicas: "esquema" (usada para la creación del boosterpack de las prácticas) y "TFG" (creada para este trabajo). La única huella que tiene esta última librería es la SOD_523F, que no es más que la SOD_523 modificada para los diodos D1, D2 y D3, pues es la que pide el fabricante.

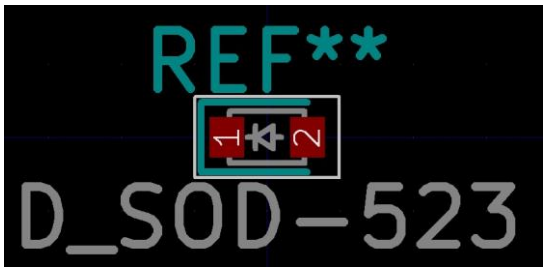

Figura 3-18. Huella de KiCad: SOD-523.

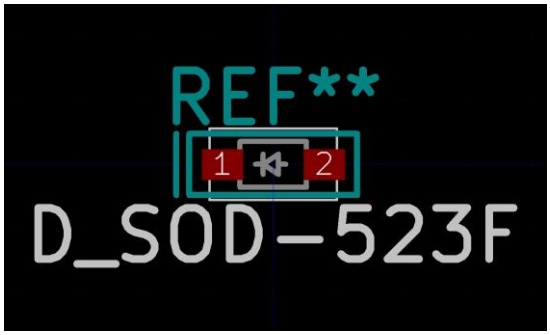

Figura 3-19. Huella de KiCad: SOD-523F.

Symbol : Footprint Assignments

4	C4	C-EUC1206K	esquema:C1206K
5	D1	D	TFG:D_SOD-523F
6	D2	D	TFG:D_SOD-523F
7	D3	D	TFG:D_SOD-523F
8	D4	1N4007	Diode_THT:D_DO-41_SOD81_P10.16mm_Horizontal
9	J1	Screw_Terminal_01x02	TerminalBlock:TerminalBlock_Altech_AK300-2_P5.00mm
10	JP1	PINHD-1X2	esquema:1X02
11	JP2	PINHD-1X2	esquema:1X02
12	JP3	PINHD-1X2	esquema:1X02
13	JP4	PINHD-1X1	esquema:1X01
14	JP5	PINHD-1X1	esquema:1X01
15	JP6	PINHD-1X1	esquema:1X01
16	JP7	PINHD-1X1	esquema:1X01
17	JP8	PINHD-1X3	esquema:1X03
18	JP9	PINHD-1X3	esquema:1X03
19	JP10	PINHD-1X2	esquema:1X02
20	JP11	PINHD-1X2	esquema:1X02
21	JP12	PINHD-1X3	esquema:1X03
22	JP13	PINHD-1X3	esquema:1X03
23	JP14	PINHD-1X2	esquema:1X02
24	JP15	PINHD-1X3	esquema:1X03
25	LED1	LED3MM	esquema:LED3MM
26	LED2	LED3MM	esquema:LED3MM
27	Q1	PBSS4120T	Package_TO_SOT_SMD:SOT-23
28	Q2	PBSS4120T	Package_TO_SOT_SMD:SOT-23
29	Q3	PBSS4120T	Package_TO_SOT_SMD:SOT-23
30	R1	R-EU_R1206	esquema:R1206
31	R2	R-EU_R1206	esquema:R1206
32	R3	3RP_1610N	esquema:3RP_1610N
33	R4	3RP_1610N	esquema:3RP_1610N
34	R5	R-EU_R1206	esquema:R1206
35	R6	R-EU_R1206	esquema:R1206
36	R7	R-EU_R1206	esquema:R1206
37	R8	R-EU_R1206	esquema:R1206
38	R9	R-EU_R1206	esquema:R1206
39	R10	R-EU_R1206	esquema:R1206
40	U1	STELLARIS-LAUNCHPADSTELLARIS-LAUNCHPAD-XL	esquema:LAUNCHPAD-XL
41	U2	MCP2562	esquema:DIP8

Figura 3-20. Listado de huellas.

3.2.3 PCB

Una vez terminado el esquema y asignadas las huellas a cada componente, se pasa a formar el PCB ("Printed Circuit Board"), es decir, el archivo que se usa para fabricar la placa con el circuito impreso. A diferencia del esquema, aquí sí se debe tener en cuenta la posición, forma y medidas de los elementos que conforman la placa. Además de colocar los elementos dentro de un espacio delimitado que hace de bordes de la placa, hay que conectarlos mediante pistas, que pueden ir por la cara superior o inferior, pero sin cruzarse en la misma. Es posible realizar esta tarea con un enrutador automático, pero por estética y por trabajar con distintos tamaños de pista se ha realizado a mano. En las siguientes figuras se muestra el PCB sin y con las pistas:

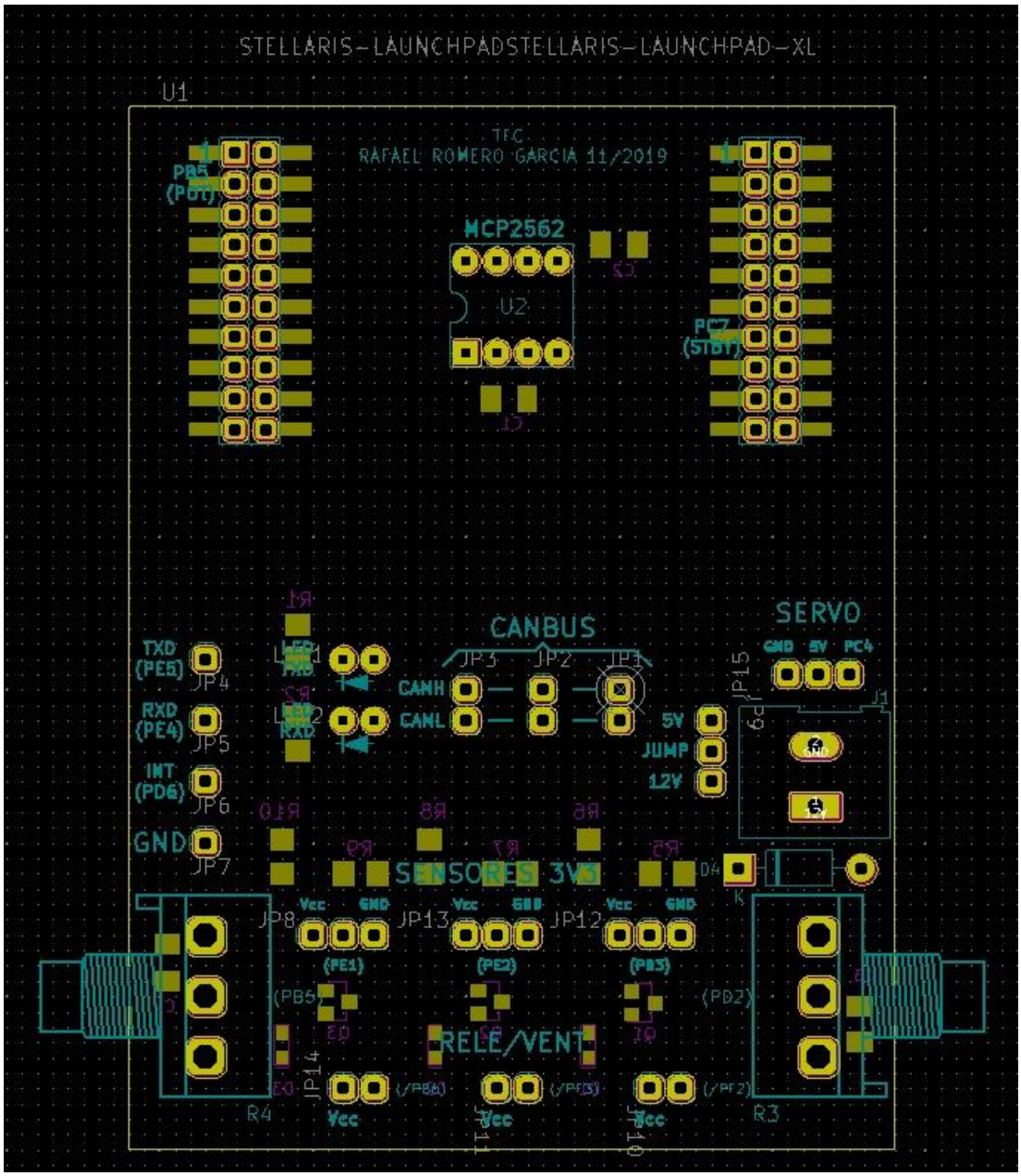

Figura 3-21. PCB de KiCad: Elementos.

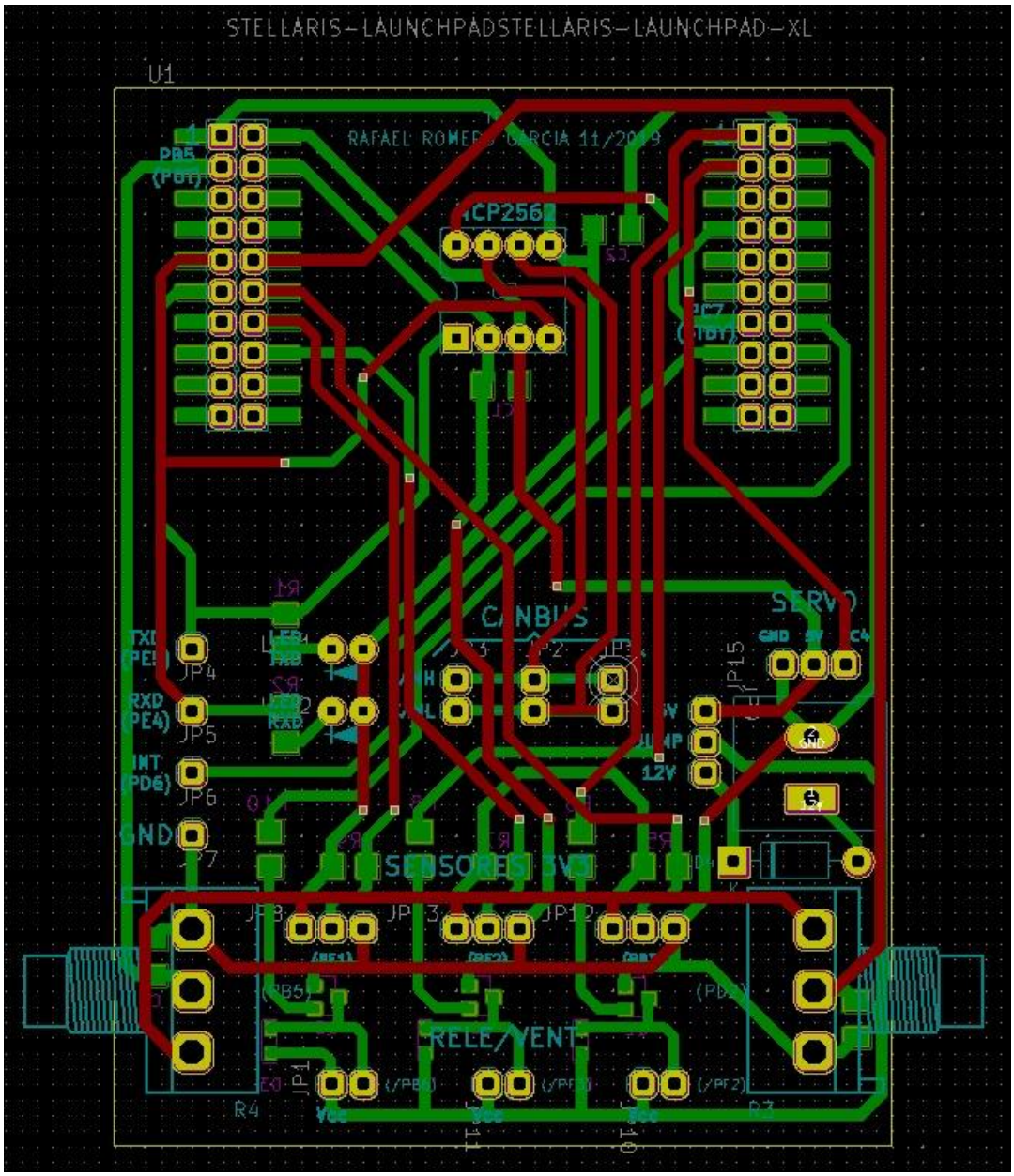

Figura 3-22. PCB de KiCad: Elementos y pistas.

Las letras y dibujos en azul son inscripciones que aparecerán en la capa superior de la placa, al igual que las moradas en la inferior.

Nótese la concentración de huellas en la parte media inferior respecto a la superior. Esto se debe a que la tarjeta con el microcontrolador va conectada a este PCB por encima mediante las 4 columnas de conectores metálicos, por lo que para operarlo no se podía dejar ningún elemento en la parte “tapada”, a excepción del transceptor CAN, que no supone ningún problema. La distribución física de todo lo mencionado aquí se puede observar mejor en las figuras incluidas en el apartado del hardware.

3.2.4 Fabricación

Una vez acabado el PCB, se crean los archivos "gerbers", que se envían al fabricante.

En la pestaña "Archivo" se selecciona "trazar".

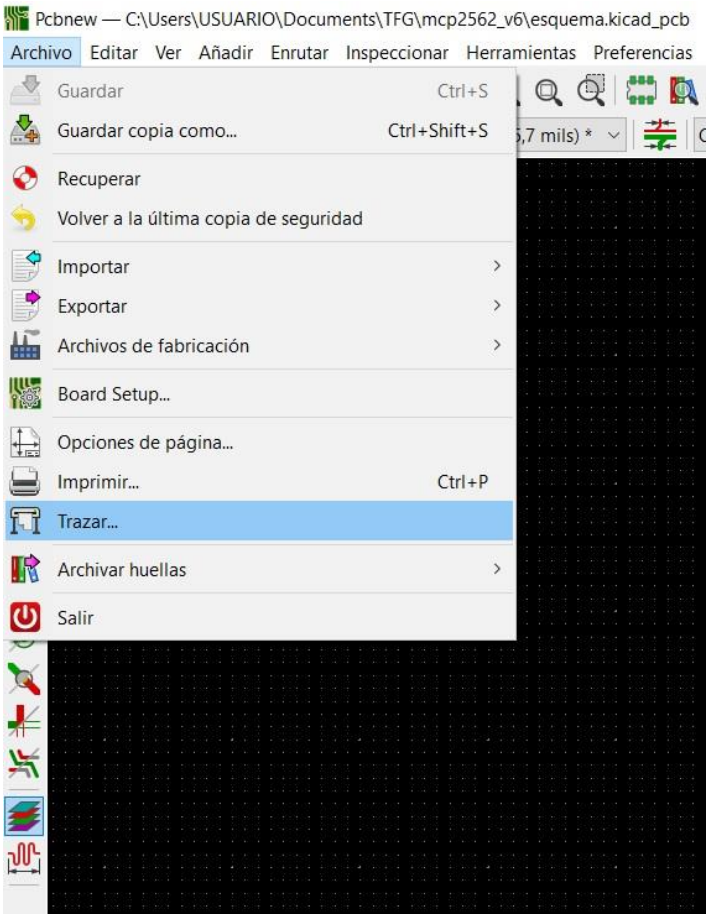

Figura 3-23. PCB de KiCad: trazar.

Se abre una ventana. Se seleccionan y completan las casillas necesarias (mirar figura siguiente), se ejecuta el DRC ("Design Rules Check") para comprobar que no se incumple ninguna regla de diseño y se genera el archivo de taladrado.

Una vez hecho esto, se pincha en trazar y se crean los archivos en la carpeta “gerbers”.

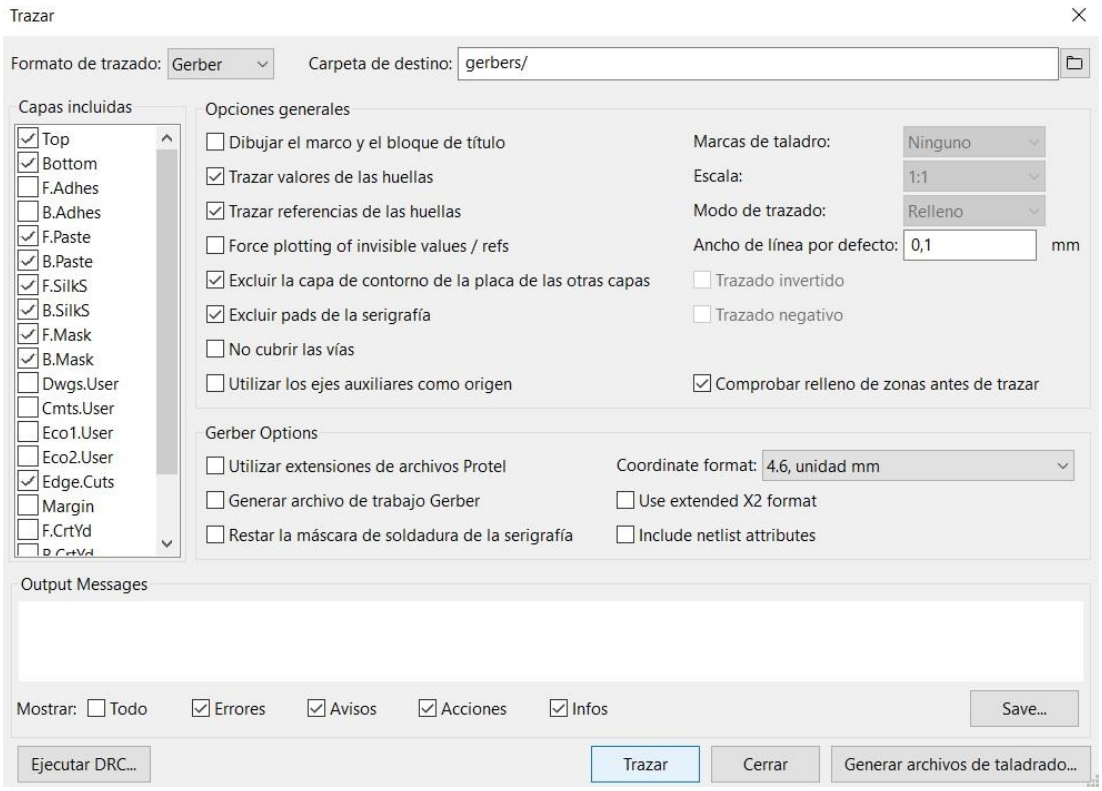

Figura 3-24. PCB de KiCad: opciones de trazar.

Una vez generados, se envían al fabricante. Para este proyecto, los archivos se enviaron a “AllPCB”.

4 DESARROLLO

El resultado esperado era el descrito en los objetivos del proyecto: las placas con los sensores captaban la temperatura, la mandaban a la placa central, esta convertía la temperatura en un determinado valor de pwm y enviaba ese valor de nuevo a los nodos para que el ventilador girara con más o menos potencia.

Después de un primer acercamiento a la tarjeta Tiva y a la programación en CCS, se procedió al trabajo con KiCad, ya detallado anteriormente, el cual no resultó muy problemático más allá de lo tedioso del proceso en cuanto a la estética del esquema, selección de componentes, huellas, distribución espacial en el PCB, dibujo de las pistas (lo que supone la mayor parte del tiempo de esta parte del proyecto), etc. Una vez se realizó esto y se tenía posesión de todos los componentes y placas necesarios para trabajar, se procedió al desarrollo del código, que ha sido la parte fundamental del trabajo en cuanto a tiempo y esfuerzo.

El primer problema que se dio en la implementación del código fue relativo al sensor de temperatura. La tarjeta que se utiliza en el trabajo y su microcontrolador no cuentan con módulo 1-wire, como sí tienen otras de la misma empresa, como algunas de la familia Tiva C TM4C129 o la MSP430. Esto dificulta mucho la tarea, ya que al no contar con el soporte (funciones, librerías, etc), hay que crear todos los archivos necesarios si se quiere trabajar con el sensor DS18B20. Finalmente, se acabó reutilizando código encontrado en internet de un usuario [1] que había creado las librerías para una tarjeta de la familia TM4C129 que tampoco contaba con el módulo. Tras unos pequeños ajustes relativos a los pines que se utilizaban, el código se pudo usar para esta tarjeta. Sin embargo, estaba escrito para "Energia", un entorno de programación de Texas Instruments que simula al de Arduino. A pesar de que CCS cuenta con una opción para importar proyectos de Energia, esta no era una opción válida ya que estos proyectos usan un compilador distinto y la fusión con el código de CANbus daba muchos problemas. Finalmente se acabó pasando el código de Energia a CCS, pero sin importarlo, es decir, incluyéndolo en el proyecto que ya contaba con la parte de CANbus sin cambiar la configuración del mismo. Se recorrió el código, cambiando las líneas que no fueran compatibles y creando nuevas librerías y funciones hasta que su implementación dejó de dar errores.

La comunicación con el sensor, como ya se ha demostrado en la parte del software, requiere de unos tiempos específicos que, aunque son del orden de los micro y milisegundos, producen un tiempo extra entre las comunicaciones entre las placas. Aunque se había conseguido implementar el código relativo al sensor en el proyecto, se produjeron grandes contratiempos en cuanto a los delays usados en la comunicación con el sensor, en el sentido de que a pesar de que el código ordenase hacer una espera específica, no se estaba produciendo exactamente dicha espera. Al principio, para producir los distintos delays se usaban dos funciones, "delayMilliseconds" y "delayMicroseconds". Ambas funciones eran en esencia iguales, la única diferencia era que la de los microsegundos tenía un divisor mil veces mayor que la de los milisegundos, pues 1 microsegundo es una milésima parte de un milisegundo.

```
void delayMS(int ms)
{
   SysCtlDelay( (SysCtlClockGet()/(3*1000))*ms ) ;
}

void delayUS(int us)
{
    SysCtlDelay( (SysCtlClockGet()/(3*1000000L))*us ) ;
}
```

La función "SysCtlDelay" hace un delay mediante un bucle de 3 instrucciones, de ahí que el argumento sea la frecuencia del sistema (proporcionada por "SysCtlClockGet") dividida entre 3. También se divide entre 1.000 o 1.000.000 según sea el delay y todo eso se multiplica por el valor del tiempo que se quiere esperar (variables ms y us).

Por ejemplo, si queremos un delay de 10 milisegundos:

$$\frac{80x10^6}{3000}x10 = \frac{80x10^4}{3}\ bucles$$

$$\frac{80x10^4}{3} bucles\ x\ 3\ instrucciones\ x \frac{1}{80x10^6} segundos\ por\ instrucción = 0.01\ segundos = 10\ milisegundos$$

La función hacía el delay exacto para el caso de los milisegundos, pero no era precisa para los microsegundos y al hacer debug para comprobar el funcionamiento, el nodo no detectaba el sensor. Se hizo la prueba en un laboratorio con un osciloscopio para comprobar que los tiempos estaban siendo los requeridos y se comprobó que no estaba siendo así. El tiempo que se indicaba esperar con la función estaba siendo otro, de ahí que la comunicación con el sensor no fuese correcta y este no estuviera siendo detectado.

Al final, se llegó a una solución poco estética pero que dio resultado: para cada delay de microsegundos que se necesita en la comunicación del sensor se estableció una función específica con el número de bucles exactos que se debían ejecutar para obtener dicho delay, los cuales no se correspondían con los que se suponían necesarios según los cálculos mencionados anteriormente. Para ello, se fue iterando con distintos valores para la cantidad de bucles hasta dar con el que proporcionaba el delay exacto en la pantalla del osciloscopio. Por ejemplo, para el delay de 10 microsegundos, cuyo argumento de la función debería ser 800/3 = 266.67, el argumento utilizado es 227:

```
void delay10US(void)
{
    SysCtlDelay( 227 ) ;
}
```

Después de solucionar este problema, el nodo detectaba el sensor y este proporcionaba la temperatura, pero a la hora de conectarla con el master mediante el bus CAN, la comunicación era muy lenta, del orden de segundos. El nodo ejecuta en cada iteración del bucle infinito del main la función loop. Dicha función añadía un tiempo de poco más de 3 segundos antes de ejecutarse la parte del código relativa a la comunicación con el CAN (descartando varias interrupciones del timer), por lo que había una diferencia entre la frecuencia con la que el master transmite información y la frecuencia con la que lo hacen los nodos, que se traducía en una espera de varios segundos entre cada actualización de la temperatura.

Para solucionar esto, se hizo uso del comando “switch(estado)-case”, es decir, se implementó en la función loop un proceso como una máquina de estados, de modo que el programa entrase y saliese de esta función pasando solo por el estado que le correspondiese en cada iteración, en lugar de ejecutar la función entera en una secuencia lineal. Así, la función solo requería de unos microsegundos en cada iteración y no perjudicaba a la comunicación entre nodo y master.

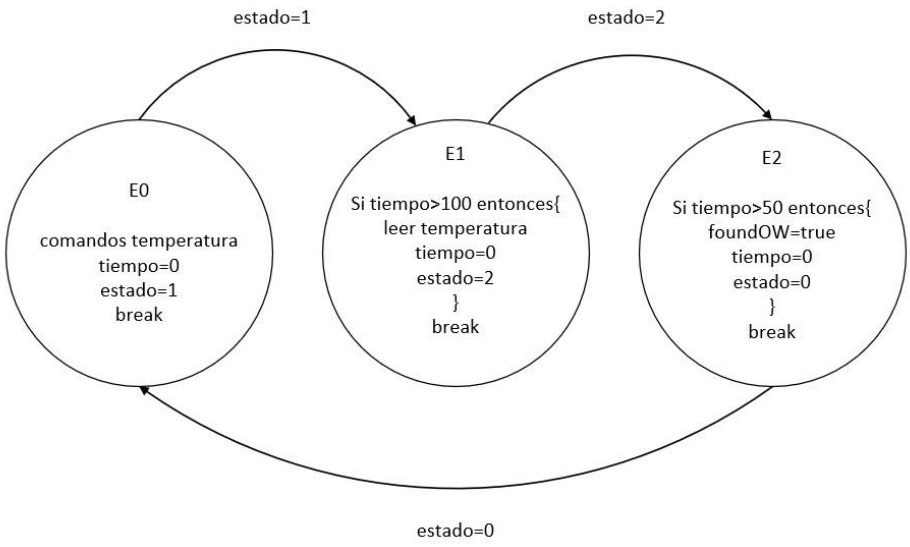

Figura 4-1. Máquina de estados en la función “loop”.

En la primera iteración se entra en el estado 0, donde se ejecutan los comandos para la lectura de la temperatura y se inicializa la variable “tiempo” a 0, que es incrementada en el handler del timer0 en cada interrupción (una cada 10 ms) de dicho timer. No se pasa directamente al siguiente estado aunque la variable “estado” valga 1, si no que se hace un “break” para que se ejecute el resto del código del main y no tener el problema de antes. Así, cada vez que se entre en el estado 1 o 2 pero no se haya producido el delay necesario en cada uno para ejecutarse sus acciones, se hace un break y se sigue con el código. Se logra que se produzcan los delays, controlados por el timer, sin que entorpezcan la ejecución del resto del programa, consiguiendo una comunicación rápida entre master y nodos.

Después de esto y de añadir al código los comentarios necesarios para una mejor comprensión de este, se pudo dar por finalizado el desarrollo del trabajo con las placas.

5 RESULTADOS

Después de solucionar todos los contratiempos, los dispositivos se comunican correctamente y el objetivo buscado desde un principio es alcanzado con éxito. Este no es un trabajo que busque uno o varios resultados numéricos en concreto, si no que el funcionamiento de los dispositivos se corresponda con los datos y variables que se manejan en el proceso (temperatura, valor de los pwm, etc) y que estos sean coherentes con lo que representan. No se ha trabajado con salida de información por pantalla ya que gracias a que CCS cuenta con la herramienta para el debugging se pueden rastrear variables.

Para el caso de los nodos se tiene:

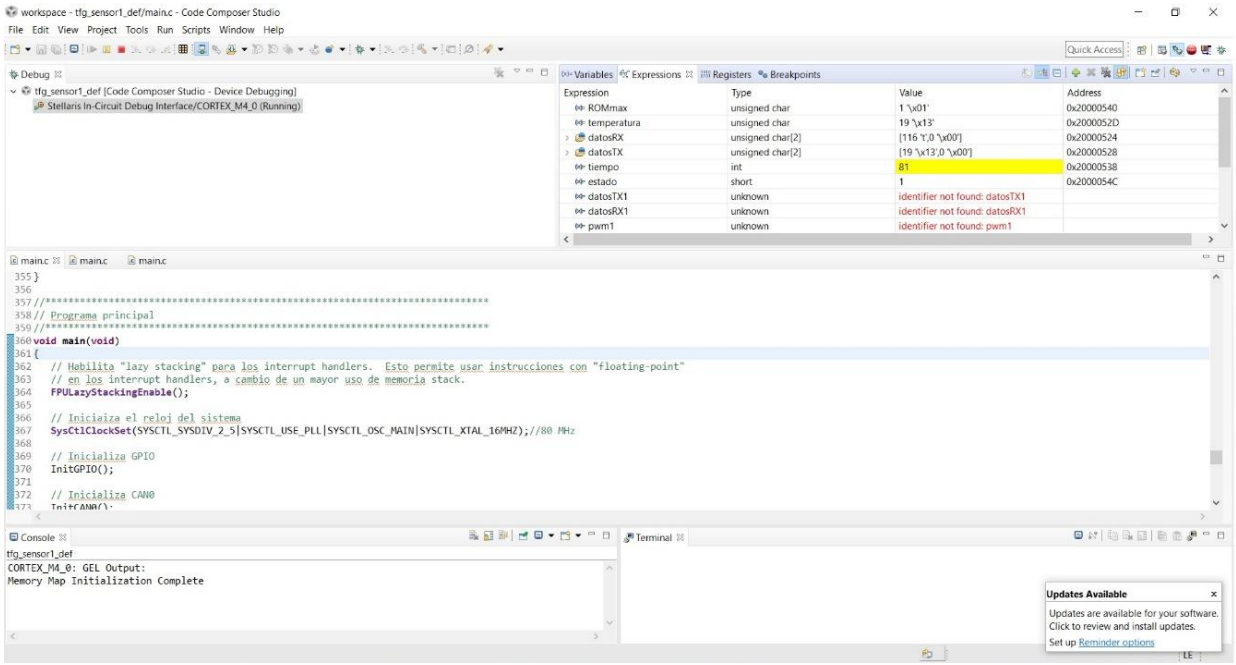

Figura 5-1. Debug nodo.

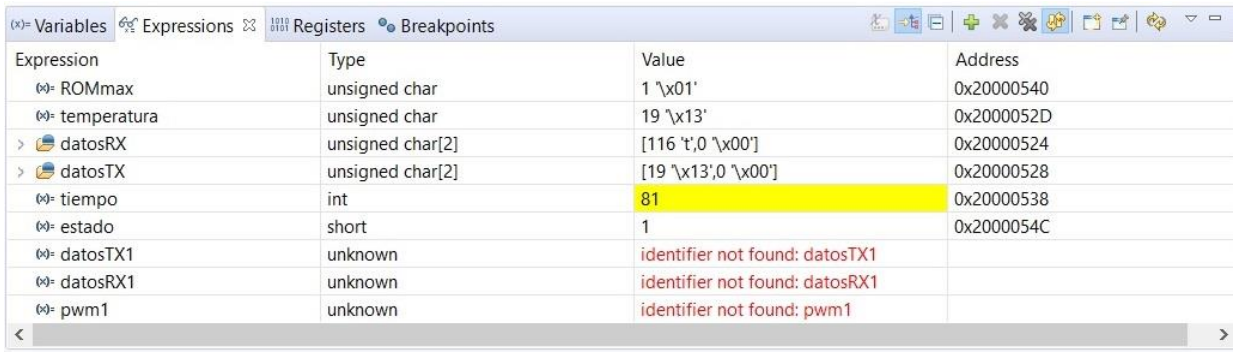

Figura 5-2. Seguimiento de variables del nodo.

"ROMmax" es una variable cuyo valor indica el número de sensores conectados al 1-wire, en este caso solo uno. La función "loop" almacena los grados centígrados en la variable "temperatura", que será la que se guarde en el array "datosTX", utilizado para transmitir información por el bus CAN. "DatosRX" es el valor pwm que llega desde el master en función de la temperatura, de ahí ese valor (19ºC en un rango de 10 a 30 para la temperatura y de 1 a 255 para el pwm le corresponde un valor de 116). "Tiempo" y "estado" son las variables usadas en "loop" para la máquina de estados.

Para el nodo 2 el caso es el mismo.

En el caso del programa del master:

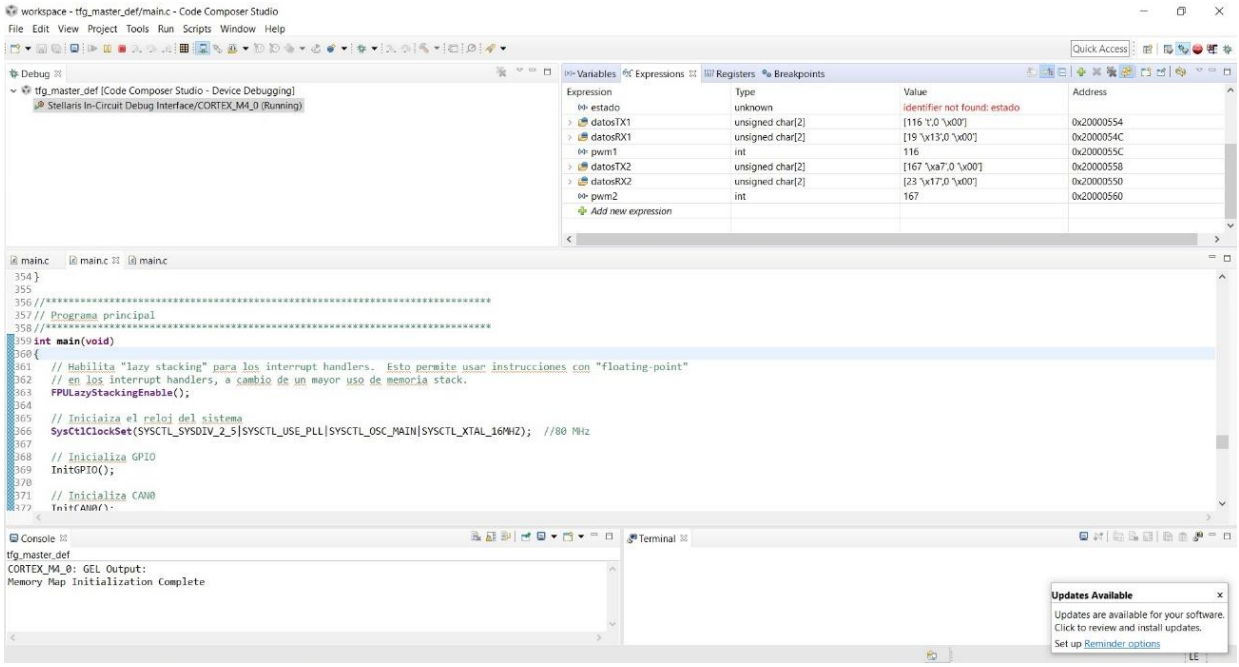

Figura 5-3. Debug master.

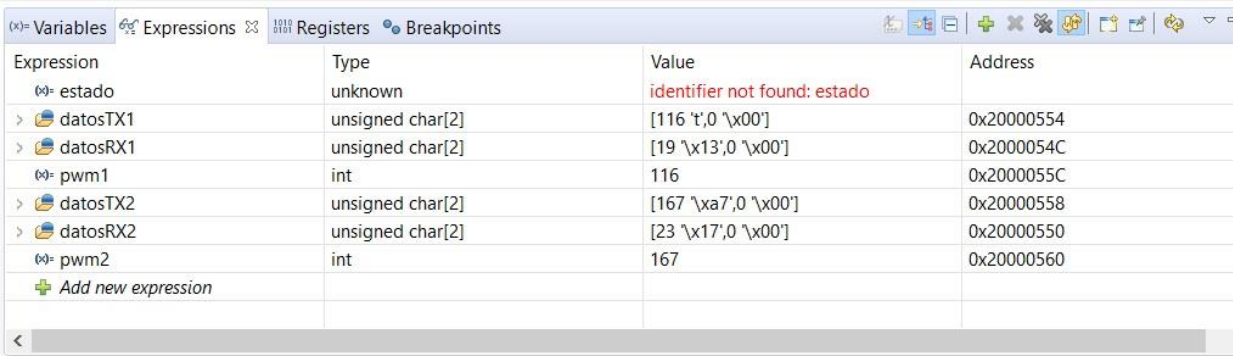

Figura 5-4. Seguimiento de variables del master.

En este caso, al haber comunicación con ambos nodos, tenemos "datosRX1" y "datosTX1", que son, respectivamente, la temperatura que llega del nodo 1 y el valor de "pwm1" que el master le envía en función de esta; y de forma análoga, "datosRX2" y "datosTX2".

Se observa la correspondencia del RX del master con el TX del nodo y viceversa.

6 CONCLUSIÓN Y POSIBLES EXTENSIONES

Después de todo el trabajo, se cumple los objetivos propuestos al principio de este documento. El hecho de haber enfrentado las dificultades explicadas anteriormente ha servido para un mayor aprendizaje acerca del mundo de la programación orientada a dispositivos electrónicos, así como de todo lo relacionado con el hardware de estos, funcionalidades, empresas, familias de dispositivos, componentes, etc.

Aun así, la principal conclusión que puede extraerse de este trabajo es que, aunque como se ha demostrado, es factible, la elección del material para llevarlo a cabo no es la idónea. Hay otras placas, no necesariamente de Texas Instruments, que cuentan tanto con drivers para el CAN como para el 1-wire, lo cual facilita mucho el trabajo, pues no se tendrían que crear manualmente los archivos y funciones relativos a la comunicación con el sensor.

Se plantea como posible alternativa al desarrollo del trabajo el uso de una de esas placas. Para ello, habría que cambiar el código por varios motivos. El uso de una placa distinta supone, entre otras cosas, que la posición y funcionalidad de los pines cambie, que las funciones utilizadas no sean las mismas y que, si se cuenta con driver para el 1-wire, no hagan falta archivos extra para la comunicación con el sensor. Si la placa escogida es de la familia Tiva C, como lo es la TM4C129, estos cambios no serán muy significativos, pero si se escoge una placa de otra familia sí es posible que el trabajo no sea tan parecido, aunque en cualquiera de los dos casos, si se cuenta con driver para el sensor, el trabajo se verá acortado en gran medida. Además, hay que tener en cuenta que, en algunos casos, debido al cambio en la distribución de los conectores de la placa, se tendría que volver a diseñar un PCB que se adapte a estos, pudiéndose partir del utilizado en este trabajo.

Otra alternativa puede ser escoger otro tipo de comunicación distinta a CANbus u otro tipo de sensor que no necesite de 1-wire. La propia placa del trabajo cuenta con driver para el protocolo de comunicación I2C, para el cual hay varios sensores de temperatura en el mercado a un precio y características similares al usado en este proyecto.

También se puede intentar mejorar el código, en especial la parte de los delays usados en la comunicación con el sensor.

Por último, el trabajo realizado se puede implementar en futuras prácticas del máster (como ya se había dicho en uno de los objetivos del proyecto). Los alumnos, que antes solo se iniciaban en la comunicación mediante el bus CAN, pueden aprender más acerca del funcionamiento de este o de otro sensor o simplemente indagar más en el uso de la placa y la codificación en CCS. También pueden trabajar con el KiCad, por ejemplo, con una práctica en la que partan de un PCB y tengan que añadirle nuevos componentes para alguna nueva funcionalidad.

REFERENCIAS

[1] Usuario "jeffman54" en https://forum.43oh.com/topic/3314-energia-library-onewire-ds18b20-430-stellaris/page/3/#comments.

BIBLIOGRAFÍA

Datasheets de los dispositivos:

Microcontrolador TM4C123GH6PM: http://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf

LaunchPad TM4C123G: http://www.ti.com/lit/ug/spmu296/spmu296.pdf

Guía de drivers: https://www.ti.com/lit/ug/spmu298d/spmu298d.pdf

Transceptor MCP2562: http://ww1.microchip.com/downloads/en/devicedoc/20005167c.pdf

CAN: http://www.ti.com/lit/an/sloa101b/sloa101b.pdf

Sensor de temperatura DS18B20: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

Sitios web:

http://www.exa.unicen.edu.ar/catedras/tmicrocon/Material/1_introduccion_a_los_ucontroladores.pdf

https://en.wikichip.org/wiki/ti/tms1000

https://aprendiendoarduino.wordpress.com/2015/03/29/microcontrolador-vs-microprocesador/

https://en.wikichip.org/wiki/microcontroller

https://www.academia.edu/11682440/Apuntes_Perif%C3%A9ricos_Tema_1_Microcontroladores_y_Procesadores_Digitales_de_Se%C3%B1al

https://www.can-cia.org/can-knowledge/

http://www.ti.com/tool/CCSTUDIO#descriptionArea

http://docs.kicad-pcb.org/5.1.5/en/kicad/kicad.html

https://www.maximintegrated.com/en/design/technical-documents/app-notes/1/162.html

https://es.wikipedia.org/wiki/Microcontrolador

ANEXO: CÓDIGO DE EJEMPLO (MAIN DEL NODO)

```
//*****************************************************************************
// Código del nodo 1
//*****************************************************************************

// Librerías utilizadas
#include <stdint.h>
#include <stdbool.h>
#include "driverlib/rom_map.h"
#include "inc/hw_can.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "driverlib/rom.h"
#include "driverlib/fpu.h"
#include "driverlib/can.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "driverlib/interrupt.h"
#include "driverlib/adc.h"
#include "driverlib/pwm.h"
#include "defines.h"

// Declaración de funciones para la comunicación con el sensor
void setup(void);
void loop(void);

// Variable a enviar al master
uint8_t temperatura;

// Variable para mostrar las interrupciones
uint8_t conmuta_int=0;

// Bandera de interrupción del Timer0
volatile bool flag_Timer0=false;

// Banderas para indicar que se ha recibido o transmitido un mensaje
volatile bool flag_RX=false;
volatile bool flag_TX=false;

// Variable donde se guardan los errores
volatile uint32_t flags_errores = 0;


// Estructuras que contienen los mensajes CAN de recepción y transmisión (RX y TX)
tCANMsgObject CAN0RxMessage;
tCANMsgObject CAN0TxMessage;

// Vetores para contener los datos para enviar o recibir.
uint8_t datosTX[2];
uint8_t datosRX[2];
```

```
// Definición de macros para la comunicación
#define NODO_ID 2

#define ID_1 1
#define ID_2 2
#define ID_3 3
#define ID_4 4

// Definición de macros para la creación de los objetos de recepción y transmisión
#if NODO_ID == 1
#define CAN0RXID1               ID_1
#define CAN0TXID1               ID_2
#define CAN0RXID1_MASK          7
#define RXOBJECT1               1
#define TXOBJECT1               2
#define CAN0RXID2               ID_3
#define CAN0TXID2               ID_4
#define CAN0RXID2_MASK          7
#define RXOBJECT2               3
#define TXOBJECT2               4
#elif NODO_ID == 2
#define CAN0RXID                ID_2
#define CAN0TXID                ID_1
#define CAN0RXID_MASK           7
#define RXOBJECT                2
#define TXOBJECT                1
#elif NODO_ID == 3
#define CAN0RXID                ID_4
#define CAN0TXID                ID_3
#define CAN0RXID_MASK           7
#define RXOBJECT                4
#define TXOBJECT                3
#else
#error NODO_ID NO DEFINIDO!!!
#endif

//*****************************************************************************
// Inicializa GPIO.
//*****************************************************************************
void InitGPIO(void)
{
    // Habilita el puerto C y espera a que se pueda usar
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOC))
    {}

    // Configura como salida el pin C7 para habilitar el driver MCP2562
    GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_7);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0); // Enable activo a nivel bajo

    // Habilita el puerto D y espera a que se pueda usar
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOD))
    {}

    // Configura como salida el pin D6 (INT) para mostrar la temporización de la
    //interrupción
    GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_6);
    GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, 0); // Inicialmente a 0
}
```

```
//*****************************************************************************
// Inicializa Timer0.
//*****************************************************************************
void InitTimer0(void)
{
    // Habilita el Timer0 y espera a que se pueda usar
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_TIMER0))
    {}

    // Definición del tiempo entre interrupciones del Timer0
    #define TIEMPO_INT_T0 (10.e-3) // 10 ms

    // Configuración del timer
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
    TimerLoadSet(TIMER0_BASE, TIMER_A, TIEMPO_INT_T0*SysCtlClockGet());

    // Habilita las interrupciones
    IntEnable(INT_TIMER0A);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Habilita el Timer0A
    TimerEnable(TIMER0_BASE, TIMER_A);
}

//*****************************************************************************
// Rutina de interrupción asociada al Timer0A
//*****************************************************************************
int tiempo=0,tiempo_timer=0;
void Timer0IntHandler(void)
{
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // Baja la bandera de la
    interrupción

    tiempo_timer++;                                 // Incrementa la variable con la
    que se controlan las interupciones

    if(tiempo_timer>10)                             // if para subir la bandera de
    interrupción que se controla en el main() cada 100 ms
    {
        tiempo_timer=0;                             // Se reinicia el conteo
        flag_Timer0=true;// Activa la bandera que se gestiona en el bucle infinito
    }
    tiempo++;//variable usada en loop() para controlar el paso de un estado a otro
}

//*****************************************************************************
// Configura CAN0
//*****************************************************************************
void InitCAN0(void)
{
    // Se usan los pines E4 y E5 para el CAN0
    // Habilita el puerto E y espera a que se pueda usar
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOE))
    {}

    // Configura las GPIO para seleccionar CAN0 en los pines PE4 y PE5.
    GPIOPinConfigure(GPIO_PE4_CAN0RX);
    GPIOPinConfigure(GPIO_PE5_CAN0TX);
```

```
    // Habilita los pines como tipo CAN
    GPIOPinTypeCAN(GPIO_PORTE_BASE, GPIO_PIN_4 | GPIO_PIN_5);

    // Habilita el periférico CAN0 y espera a que se pueda usar
    SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN0);
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_CAN0))
    {}

    // Inicializa el controlador CAN
    CANInit(CAN0_BASE);

    // Configura el CAN a 500 kHz.
    CANBitRateSet(CAN0_BASE, SysCtlClockGet(), 500000);

    // Habilita las interrupciones del módulo CAN
    CANIntEnable(CAN0_BASE, CAN_INT_MASTER | CAN_INT_ERROR | CAN_INT_STATUS);

    // Habilita la interrupción en el registro NVIC (máscara de interrupciones
    activas)
    IntEnable(INT_CAN0);

    // Habilita el CAN
    CANEnable(CAN0_BASE);

   // Initcializa el objeto de recepción RXOBJECT
   CAN0RxMessage.ui32MsgID = CAN0RXID;
   CAN0RxMessage.ui32MsgIDMask = CAN0RXID_MASK;
   CAN0RxMessage.ui32Flags = MSG_OBJ_RX_INT_ENABLE | MSG_OBJ_USE_ID_FILTER;
   CAN0RxMessage.ui32MsgLen = sizeof(datosRX);
   CANMessageSet(CAN0_BASE, RXOBJECT, &CAN0RxMessage, MSG_OBJ_TYPE_RX);

   // Initcializa el objeto de transmisión TXOBJECT
   datosTX[0] = 0;
   datosTX[1] = 0;
   CAN0TxMessage.ui32MsgID = CAN0TXID;
   CAN0TxMessage.ui32MsgIDMask = 0;
   CAN0TxMessage.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
   CAN0TxMessage.ui32MsgLen = sizeof(datosTX);
   CAN0TxMessage.pui8MsgData = (uint8_t *)&datosTX;

   // Deshabilita reintentos de transmisión si hay errores
   CANRetrySet(CAN0_BASE, false);
}

//*****************************************************************************
// CAN0 Interrupt Handler. Chequea la fuente de interrupción.
//*****************************************************************************
void CAN0IntHandler(void)
{
   uint32_t status;

   // Valores posibles del registro CAN_INT_STS_CAUSE
   // 0      = No hay interrupción pendiente
   // 1-32   = Número de objeto que ha provocado la interrupción
   // 0x8000 = Status interrupt (CAN_INT_INTID_STATUS)
   status = CANIntStatus(CAN0_BASE, CAN_INT_STS_CAUSE);
```

```
    if(status == CAN_INT_INTID_STATUS)
    {
        // Lee el estado del controlador para obtener los errores que se hayan
        producido
        uint32_t controllerStatus = CANStatusGet(CAN0_BASE, CAN_STS_CONTROL);
        // Añadir a flags_errores para su posterior gestión
        flags_errores |= controllerStatus;
    }
    else if(status == RXOBJECT) // Se ha recibido un objeto RXOBJECT
    {
        CANIntClear(CAN0_BASE, RXOBJECT);    // Baja la bandera de la interrupción
        flag_RX = true;                      // Activa la bandera que se gestiona en
        el bucle infinito
        flags_errores = 0;
    }
    else if(status == TXOBJECT) // Se ha recibido un objeto TXOBJECT
    {
        CANIntClear(CAN0_BASE, TXOBJECT);    // Baja la bandera de la interrupción
        flag_TX = true;                      // Activa la bandera que se gestiona en
        el bucle infinito
        flags_errores = 0;
    }
    else
    {
        // Otras fuentes de interrupción
    }
}

//*****************************************************************************
// Gestión de errores del CAN
//*****************************************************************************
void CANErrorHandler(void)
{
    // Se entra en un if u otro en funcion del error y se gestiona

    if(flags_errores & CAN_STATUS_BUS_OFF)
    {
        // Gestionar el error aquí
        flags_errores &= ~(CAN_STATUS_BUS_OFF);
    }
    if(flags_errores & CAN_STATUS_EWARN)
    {
        // Gestionar el error aquí
        flags_errores &= ~(CAN_STATUS_EWARN);
    }
    if(flags_errores & CAN_STATUS_EPASS)
    {
        // Gestionar el error aquí
        flags_errores &= ~(CAN_STATUS_EPASS);
    }
    if(flags_errores & CAN_STATUS_RXOK)
    {
        // Gestionar el error aquí
        flags_errores &= ~(CAN_STATUS_RXOK);
    }
    if(flags_errores & CAN_STATUS_TXOK)
    {
        // Gestionar el error aquí
        flags_errores &= ~(CAN_STATUS_TXOK);
    }
```

```
    switch(flags_errores&CAN_STATUS_LEC_MSK)
    {
        case CAN_STATUS_LEC_NONE:
            // Gestionar el error aquí
            break;
        case CAN_STATUS_LEC_STUFF:
            // Gestionar el error aquí
            break;
        case CAN_STATUS_LEC_FORM:
            // Gestionar el error aquí
            break;
        case CAN_STATUS_LEC_ACK:
            // Gestionar el error aquí
            break;
        case CAN_STATUS_LEC_BIT1:
            // Gestionar el error aquí
            break;
        case CAN_STATUS_LEC_BIT0:
            // Gestionar el error aquí
            break;
        case CAN_STATUS_LEC_CRC:
            // Gestionar el error aquí
            break;
        case CAN_STATUS_LEC_MASK:
            // Gestionar el error aquí
            break;
    }

    flags_errores &= ~(CAN_STATUS_LEC_MSK);

    // Si todavía hay algún error no gestionado anteriormente
    if(flags_errores !=0)
    {
        // Gestionar el error aquí
    }
}

//*****************************************************************************
// Inicializa el pin del ventilador
//*****************************************************************************
void InitVent(void)
{
     #define PWM_MAX2 255
     #define PWM_MIN2 1

     // Habilita el puerto F y espera a que se pueda usar
     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
     while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF))
     {}

     // Habilita el PWM1 y espera a que se pueda usar
     SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
     while(!SysCtlPeripheralReady(SYSCTL_PERIPH_PWM1))
     {}

     //Configura el predivisor de frecuencia
     SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

     //Configra el pin PF2 como PWM
     GPIOPinConfigure(GPIO_PF2_M1PWM6);
```

```
    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_2);
    //Configura el modo de funcionamiento del PWM
    PWMGenConfigure(PWM1_BASE, PWM_GEN_3, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
    PWMOutputState(PWM1_BASE, PWM_OUT_6_BIT, true);
    PWMGenPeriodSet(PWM1_BASE, PWM_GEN_3, PWM_MAX2);
    PWMPulseWidthSet(PWM1_BASE, PWM_OUT_6,PWM_MIN2);
    PWMGenEnable(PWM1_BASE, PWM_GEN_3);
}

//*****************************************************************************
// Programa principal
//*****************************************************************************
void main(void)
{
   // Habilita "lazy stacking" para los interrupt handlers.  Esto permite usar
   instrucciones con "floating-point" en los interrupt handlers, a cambio de un mayor
   uso de memoria stack.
   FPULazyStackingEnable();

   // Iniciaiza el reloj del sistema

SysCtlClockSet(SYSCTL_SYSDIV_2_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
//80 MHz

   // Inicializa GPIO
   InitGPIO();

   // Inicializa CAN0
   InitCAN0();

   // Inicializa el Timer0
   InitTimer0();

   // Iniciliza el ventilador
   InitVent();

   // Inicia comunicación con el sensor
   setup();

   // Habilita globalmente las interrupciones
   IntMasterEnable();

   // Se entra en el bucle infinito donde se produce la comunicación constante con el
   master
   while(1)
   {
      loop();     // Función que obtiene el valor de temperatura que mide el sensor

      if(flag_RX) // Se ha producido interrupción por RX (recibe del master el valor
      pwm)
      {
         flag_RX = false;

         // Apunta a los datos que se van a recibir
         CAN0RxMessage.pui8MsgData = (uint8_t *) &datosRX;

         // Guarda el mensaje recibido en el objeto RXOBJECT
         CANMessageGet(CAN0_BASE, RXOBJECT, &CAN0RxMessage, 0);
```

```
            // Guarda el dato recibido en el array datosRX en una variable de tipo
            entero para manejarla
            int valor_pwm_recibido=datosRX[0]&0xFF;
            valor_pwm_recibido|=(datosRX[1]&0xFF)<<8;

            if(valor_pwm_recibido<=255 && valor_pwm_recibido>=1) // si el valor pwm
            recibido está dentro del rango de operación establecido
                PWMPulseWidthSet(PWM1_BASE, PWM_OUT_6, valor_pwm_recibido); // establece
                el valor del pwm para el ventilador
        }
        else if(flag_TX)
        {
            flag_TX = false;
        }
        else
        {
            if(flags_errores != 0) // Si ha habido errores, se gestionan
            {
                CANErrorHandler();
            }
            if(flag_Timer0)        // Interrupción del timer
            {
                flag_Timer0=false;

                conmuta_int=!conmuta_int;

                if(conmuta_int)
                   GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, GPIO_PIN_6); // Activa pin
                 INT (boosterpack)
                else
                   GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, 0); // Desactiva pin INT

                datosTX[0]=temperatura&0xFF; // parte baja
                datosTX[1]=(temperatura>>8)&0xFF; // parte alta

                // Envía el valor de la temperatura al master
                CANMessageSet(CAN0_BASE, TXOBJECT, &CAN0TxMessage, MSG_OBJ_TYPE_TX);
            }
        }
    }
}
```