ESCUELA SUPERIOR DE INGENIERÍA
Y TECNOLOGÍA

TRABAJO DE FIN DE GRADO

DISEÑO DE UNA RED BASADA EN EL BUS CAN PARA LA ADQUISICIÓN DE DATOS

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Alumno: Jorge Pérez Barreto

Tutor: Alejandro José Ayala Alfonso

Junio 2019

Abstract

The aim of this project is focused on designing a vehicle sensor network based on CAN (Controller Area Network) technology.

This network consists of a series of nodes, each one designed to meet vehicle specific needs, such as parameters reading or light system management. There is no main master node regarding communications, since CAN bus is a message driven protocol where the most important message is transmitted through the channel.

The grid is thought to work as a broadcasting system in which every message is received and filtered by all nodes, being processed if meant to be read or being discarded otherwise.

The present project covers all the development included in node communication, topology design, sensor signal conditioning and PCB (Printed Circuit Board) making.

Índice

Capítulo I. Introducción general

Capítulo I. Introducción general

I.1. Objetivo

El presente proyecto tiene como objetivo el diseño de una red de comunicaciones basada en el bus CAN (Controller Area Network) para gestionar las interacciones entre los distintos sensores de un vehículo, en principio un coche eléctrico.

El bus CAN [1] posee una topología basada en nodos multimaestro. Cada uno de estos nodos será implementado mediante un Arduino UNO [2], encargado de recoger los datos de los sensores, procesar la información y generar los mensajes a transmitir, y un módulo MCP2515 [3], el cual traducirá el mensaje de Arduino al formato CAN y lo emitirá por dicho bus.

Algunos sensores y actuadores (como las luces de alumbrado) serán simulados, dado que el objetivo de este TFG es diseñar la red de comunicaciones, no los sensores o acondicionadores. Aun así, algunos de estos serán sensores reales de automóviles de gasolina, para demostrar la aplicabilidad real del proyecto.

En resumen, la estructura del modelo se reduce a 3 nodos, 2 de los cuales recogen medidas de los distintos sistemas del coche para transmitir su información, y el último se encarga de escuchar y almacenar los datos.

1. Nodo de medición/emisión constante
2. Nodo de gestión de luces
3. Nodo *sniffer*

I.2. Alcance

Como ya se ha mencionado, el núcleo del proyecto está constituido por el diseño de la red de comunicaciones, tanto a nivel topológico, como de programación (software) y físico (hardware).

Pese a no ser un requerimiento específico del proyecto, también se diseñaron una serie de sistemas de sensores de prueba con el fin de simular las necesidades de un coche, así como sensores reales de automóvil con su correspondiente circuito acondicionador de señal diseñado para poder leerlos con el Arduino UNO.
Entre los sensores reales empleados se encuentran 2 basados en efecto Hall para medir revoluciones/velocidad, y uno de temperatura tipo NTC.

I.3. Antecedentes

El bus CAN ha sido ampliamente utilizado desde su creación en 1983 por parte de Robert Bosch con el fin de reducir el cableado en automóviles. Hoy en día su uso se ha extendido a todos los campos de la industria, como la aviación (CANaerospace), la agricultura (ISOBUS), la industria marina (NMEA 2000), y, sobre todo, su uso en la automoción: la automatización industrial (CANopen, DeviceNet). [4]

Cabe destacar que este tipo de bus no está centrado en el control de los dispositivos o sensores que engloba, sino de su monitorización. En caso de querer controlar a nivel jerárquico el estado de los nodos sería mucho más productiva una arquitectura tipo maestro-esclavo. Esto no representa un problema para nosotros, dado que en caso de querer realizar acciones de control, cada nodo gestiona sus propios sensores y actuadores, de modo que tenemos una estructura de control no jerárquica (al menos no jerárquica entre varios Arduino, puede llegar a ser jerárquica entre un Arduino y sus propios periféricos).

En un coche típico podemos encontrar una gran variedad de sensores, siendo los más habituales los de presión y transmisión (presión, velocidad, revoluciones, posición del pedal…), los de seguridad (ABS, EBD, BAS) y los de confort (regulación de temperatura, sensor de aparcamiento por ultrasonidos).

Capítulo II. Recursos utilizados

Capítulo II. Recursos utilizados

II.1. Herramientas de laboratorio

Las herramientas de laboratorio son las usuales para el diseño y comprobación de cualquier circuito.

El osciloscopio, esencial para comprobar las señales a la salida de los sensores, y garantizar que los niveles de voltaje son los permitidos por las entradas analógicas o digitales del Arduino. El generador de señales, usado para simular las salidas de los sensores. El multímetro, empleado para medir valores constantes de voltaje o intensidad, se usa en conjunto con el osciloscopio. La protoboard, necesaria para comprobar el funcionamiento de los diseños antes de pasarlos a la PCB.

II.2. Arduino UNO

El Arduino UNO es el eje central de cada nodo del sistema. Se encarga de todo el procesamiento lógico desde el valor analógico o digital de la señal del sensor hasta su conversión en un formato apropiado para la trama del bus CAN.

Los elementos principales que lo conforman son: la fuente de alimentación, que se conecta al puerto USB del ordenador, los pines de tierra y alimentación (GND, 5V, 3.3V), los pines analógicos (A0 – A5) y los digitales (0 – 13).

Son de especial interés los pines digitales 2 y 3, ya que corresponden con las interrupciones INT0 y INT1, necesarias para medir las revoluciones (el modo de funcionamiento de cada nodo será abordado con mayor detalle en el capítulo V), y los pines digitales 10, 11, 12 y 13, ya que permiten la comunicación SPI, que es la utilizada entre el Arduino y el MCP2515. También se usan otros pines analógicos y digitales sin función específica para hacer mediciones de los sensores.

Una ventaja a señalar del Arduino UNO es que se ajusta bastante bien a nivel de pines para los requerimientos de nuestro sistema, tanto a nivel de comunicaciones (SPI) como de lectura de sensores. Además, su reducido precio y gran cantidad de información disponible sobre el mismo hace más sencillo el desarrollo.

Hay que destacar que, pese a que el Arduino UNO se usa como plataforma de desarrollo de prototipos, en la fase final en PCB sólo se utiliza su CPU, el Atmega8515, para así ahorrar espacio y optimizar el uso de pines. [2]

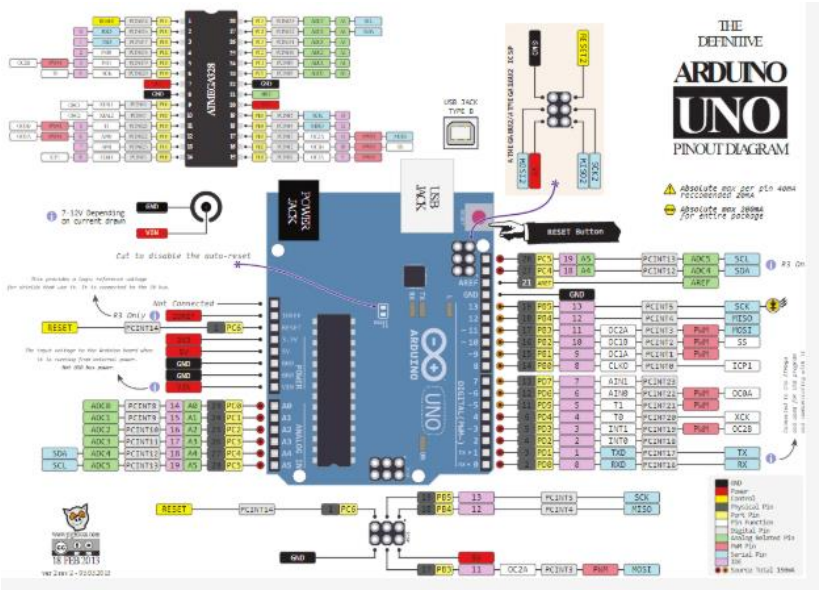

FIGURA II - 1 – ARDUINO UNO [5]

II.3. Módulo MCP2515

El MCP2515 es un módulo controlador CAN diseñado específicamente para ejercer de interfaz entre Arduino y dicho bus. Su función es la de traducir los mensajes provenientes del Arduino a la trama apropiada para poder transmitir por el bus según los requerimientos de la capa de enlace (se entrará más en profundidad en el protocolo CAN en el capítulo IV).

El módulo se comunica con el Arduino mediante protocolo SPI (Serial Peripheral Interface). Este tipo de comunicación tiene jerarquía maestro – esclavo, y generalmente se utiliza para controlar múltiples esclavos con un único maestro, aunque en nuestro caso la relación de cada Arduino con el MCP2515 es de uno a uno, de modo que no es necesario que haya varios pines de CS (Chip Select) o SS (Slave Select), como se ve en la figura II-2. [3] [6]

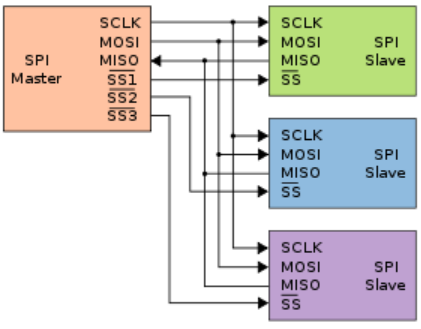

FIGURA II - 2 – COMUNICACIÓN SPI [6]

El módulo MCP2515 posee 7 pines, de los cuales sólo utilizaremos 6. Los pines SCK, SI, SO y CS son necesarios para la comunicación con Arduino, mientras que los pines Vcc y GND corresponden a alimentación y tierra, respectivamente. El pin INT se emplea para comunicarse mediante interrupciones, lo cual no ha sido necesario en el presente trabajo. [3]

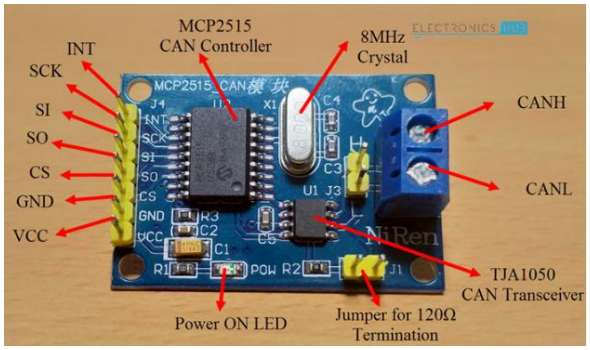

FIGURA II - 3 – MÓDULO MCP2515 [7]

II.4. Sensores de efecto Hall

Se denomina Efecto Hall a la aparición de un voltaje en un semiconductor por el que circula una corriente en presencia de un campo magnético perpendicular al movimiento de las cargas. [8]

Los sensores basados en este efecto generan pulsos de voltaje cuando se les acerca una pieza magnética. En los vehículos se suelen usar para medir revoluciones debido a la facilidad de colocar el sensor fijo en una posición cercana a la rueda, y colocar los objetos magnéticos en la propia rueda, para detectar su paso.

II.5. NTC (*Negative Temperature Coefficient)*

Los sensores tipo PTC y NTC varían su resistencia en función de la temperatura a la que se encuentran. Esto se debe a que están fabricados con materiales semiconductores, que varían su número de portadores de carga en función de la temperatura, lo que significa que habrá mayor o menor número de electrones en la capa de conducción, facilitando en mayor o menor medida el paso de corriente. Esto da lugar a una variación de la resistencia del material. La relación de la resistencia con la temperatura es de carácter exponencial.

En el caso de la NTC, al elevar la temperatura aumenta el número de portadores de carga, lo que reduce la resistencia. [9]

II.6. Librerías de SPI y CAN

Las librerías utilizadas son: la “SPI.h”, que contiene Arduino por defecto, “mcp2515.h” y “can.h”, añadidas a Arduino de forma externa para el control del módulo y las comunicaciones.

A continuación, se nombrarán brevemente las principales funciones utilizadas en las librerías y se explicará el funcionamiento de las más importantes.

II.6.1. “SPI.h”

En la librería “SPI.h” sólo se usa directamente la función “SPI.begin()” para inicialización. El resto de funciones no se usan de manera directa, sino que forman parte de las funciones de la librería “mcp2515.h”. Lo mismo ocurre para “can.h”. Ambas están incluidas en “mcp2515.h”.

```
1  #ifndef _MCP2515_H_
2  #define _MCP2515_H_
3
4  #include <SPI.h>
5  #include "can.h"
```

FIGURA II - 4 – DECLARACIÓN DE LA LIBRERÍA “MCP2515.H”

II.6.2. “can.h”

Esta librería se usa principalmente para definir el formato del mensaje en una variable tipo *struct* para poder ser usada posteriormente por “mcp2515.h”. [10]

```
39 struct can_frame {
40     canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
41     __u8    can_dlc; /* frame payload length in byte (0 .. CAN_MAX_DLEN) */
42     __u8    data[CAN_MAX_DLEN] __attribute__((aligned(8)));
43 };
```

FIGURA II - 5 – DECLARACIÓN DEL “FRAME” DEL BUS CAN

II.6.3. "mcp2515.h"

Su utilización se comentará brevemente dado que no es propia de Arduino con el fin de justificar su uso, pero no se entrará en detalle en su funcionamiento pues no entra dentro del alcance de este trabajo la gestión de los *buffers* de envío/recepción.

En la librería "mcp2515.h" todas las funciones tienen una estructura similar, ya que comienzan y finalizan con los mismos bloques, como se puede observar en el siguiente diagrama de flujo (figura II-6):

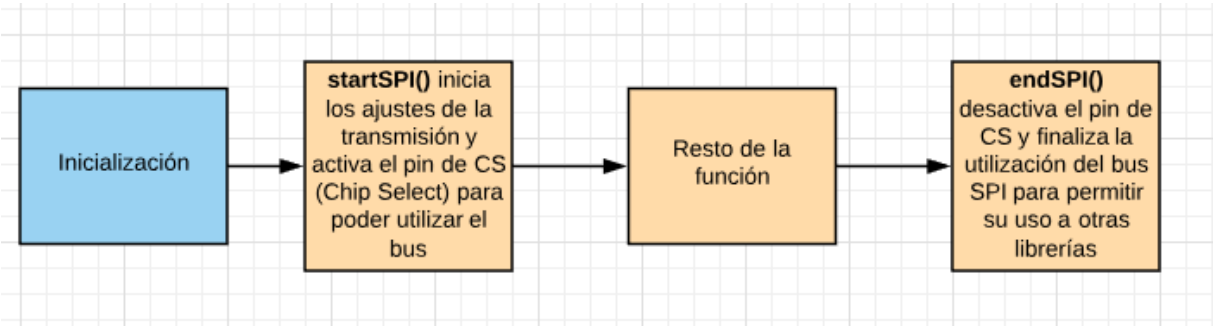

FIGURA II - 6 – DIAGRAMA DE FLUJO DE LAS FUNCIONES DE LA LIBRERÍA "MCP2515.H"

Básicamente, esta estructura inicializa la comunicación SPI, haciendo uso de las funciones startSPI() y endSPI(), que a su vez emplean funciones de la biblioteca SPI, como se mencionó antes. El comando digitalWrite(SPICS, LOW) habilita el pin 10 del Arduino UNO, correspondiente al pin CS del SPI, y después lo deshabilita al finalizar la configuración.

```
23  void MCP2515::startSPI() {
24      SPI.beginTransaction(SPISettings(SPI_CLOCK, MSBFIRST, SPI_MODE0));
25      digitalWrite(SPICS, LOW);
26  }
27
28  void MCP2515::endSPI() {
29      digitalWrite(SPICS, HIGH);
30      SPI.endTransaction();
31  }
```

FIGURA II - 7 – FUNCIONES STARTSPI() Y ENDSPI()

- **MCP2515**

El constructor es la función más importante, dado que nos permite usar los mensajes CAN como objetos de C++, así como declarar la posición del pin CS al llamar a la función.

```
14  MCP2515::MCP2515(const uint8_t _CS)
15  {
16      SPI.begin();
17
18      SPICS = _CS;
19      pinMode(SPICS, OUTPUT);
20      endSPI();
21  }
```

FIGURA II - 8 – CONSTRUCTOR DE LA CLASE MCP2515

```
MCP2515 mcp2515(10); //Pin del SS(Slave Select) del SPI
```

FIGURA II - 9 – USO DEL CONSTRUCTOR MCP2515

- **mcp2515.setBitrate()**

También es una función esencial, dado que permite regular, por medio de sus 2 parámetros de entrada, la velocidad de transmisión del bus CAN, así como sincronizar los relojes del Arduino y el MCP2515. Los valores de ambos parámetros se deben elegir entre una serie de valores predefinidos disponibles en la librería.

Respecto a la velocidad de transmisión del bus CAN, se deben cumplir una serie de requisitos establecidos en un estándar del que se hablará en el capítulo 3.

```
191  MCP2515::ERROR MCP2515::setBitrate(const CAN_SPEED canSpeed, CAN_CLOCK canClock)
192  {
193      ERROR error = setConfigMode();
194      if (error != ERROR_OK) {
195          return error;
196      }
197
198      uint8_t set, cfg1, cfg2, cfg3;
199      set = 1;
200      switch (canClock)
201      {
202          case (MCP_8MHZ):
203          switch (canSpeed)
204          {
205              case (CAN_5KBPS):                                          //   5KBPS
206              cfg1 = MCP_8MHz_5kBPS_CFG1;
207              cfg2 = MCP_8MHz_5kBPS_CFG2;
208              cfg3 = MCP_8MHz_5kBPS_CFG3;
209              break;
210
211              case (CAN_10KBPS):                                         //  10KBPS
212              cfg1 = MCP_8MHz_10kBPS_CFG1;
213              cfg2 = MCP_8MHz_10kBPS_CFG2;
214              cfg3 = MCP_8MHz_10kBPS_CFG3;
215              break;
216
217              case (CAN_20KBPS):                                         //  20KBPS
218              cfg1 = MCP_8MHz_20kBPS_CFG1;
219              cfg2 = MCP_8MHz_20kBPS_CFG2;
220              cfg3 = MCP_8MHz_20kBPS_CFG3;
221              break;
222
223              case (CAN_31K25BPS):                                       //  31.25KBPS
224              cfg1 = MCP_8MHz_31k25BPS_CFG1;
225              cfg2 = MCP_8MHz_31k25BPS_CFG2;
226              cfg3 = MCP_8MHz_31k25BPS_CFG3;
227              break;
```

FIGURA II - 10 – FUNCIÓN SETBITRATE()

- **mcp2515.reset()**
- **mcp2515.setNormalMode()**
- **mcp2515.sendMessage()**
- **mcp2515.readMessage()**

Estas 4 funciones no serán explicadas en el presente capítulo debido a que para comprender su funcionamiento es necesario entender como funciona el esquema de envío y recepción de mensajes, así como los registros asociados a los *buffers* de entrada y salida, lo cual se hará en el Capítulo IV. En este capítulo se dará una explicación de como funcionan algunas de estas funciones. [10]

II.7. Sensores

Los sensores utilizados son principalmente pertenecientes a tres categorías: sensores de revolucion, sensores de velocidad y sensores de temperatura. Desde los sensores de prueba hasta los reales se han ido escalando las versiones, por ejemplo; para los sensores de revolución el primer prototipo consistió en un LED enfrentado a un fototransistor, que era interrumpido por las aspas internas de un ventilador, para simular las interrupciones del futuro sensor Hall.

Más tarde fue sustituido el LED por un láser, para ganar en precisión y poder medir mayores valores de frecuencia. Finalmente se implementó el sensor real.

Capítulo III. Introducción al bus CAN

Capítulo III. Introducción al bus CAN

El bus CAN es un protocolo de comunicaciones diseñado por la marca alemana Bosch, cuyo objetivo era el de reducir el cableado en los automóviles. El estándar que rige este protocolo es la **ISO 11898** [1]. Generalmente sólo define las 2 primeras capas del modelo OSI (capa física y capa de enlace), pudiendo llegar a la capa de aplicación en algun protocolo específico.

Se trata de un sistema multimaestro que trabaja en modo difusión, es decir; cada mensaje enviado a través del bus llega a todos los nodos por igual.

Debido a este sistema multimaestro es necesaria una forma de decidir quién es el maestro en cada iteración del proceso, de lo contrario se producirían colisiones en el canal, y se perdería información. El bus CAN soluciona esto por medio del método de token estocástico con arbitraje por prioridad. En este método cualquier nodo puede iniciar un intento de transmisión si el bus no está ocupado. Para iniciar la transmisión se envía una trama especial que contiene un identificador de 11 bits, el cual se ha elegido en función de la prioridad del mensaje, de modo que si hay colisión entre ambos mensajes, sólo continua aquel mensaje que tenga el identificador más prioritario, que debido a las propiedades de la capa física, se trata del identificador con el número más bajo.

La **capa física** está formada por un par trenzado de 2 hilos con una impedancia característica de 120 Ω. Se tienen 2 estados lógicos: un 0 representa un estado dominante (CAN *high*) y un 1; un estado recesivo (CAN *low*). En estado recesivo, ambos cables se encuentran al mismo nivel de tensión, mientras que en estado dominante aparece entre los cables una tensión diferencial de al menos 1.5 V.

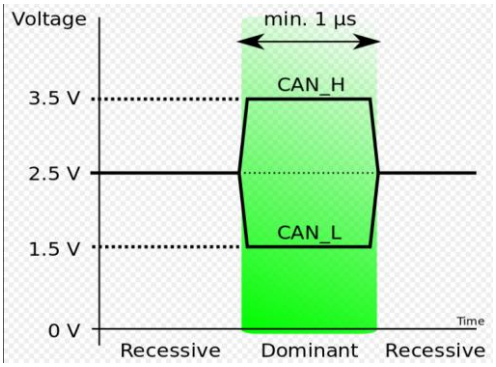

FIGURA III - 1 – ESTADOS DOMINANTE Y RECESIVO EN EL BUS [11]

Al colisionar un estado dominante con un estado recesivo en el proceso de arbitración predomina el dominante, debido a esta tensión diferencial. Esto es lo que provoca que cuando hay colisiones entre mensajes, el 0 gane al 1 en el proceso.

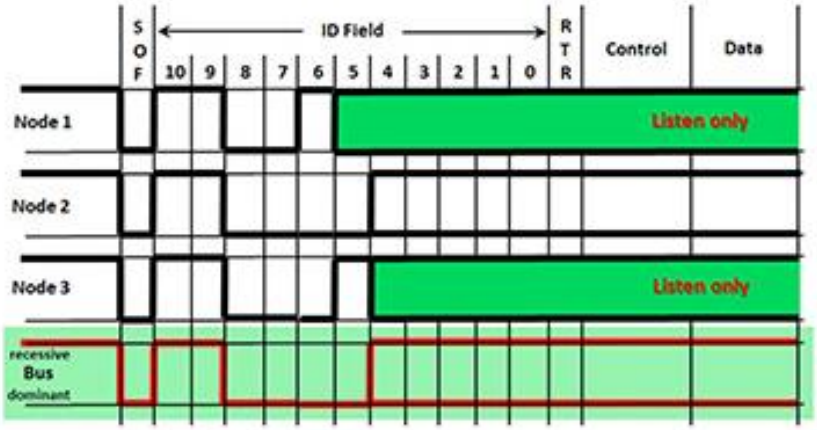

FIGURA III - 2 – PROCESO DE ARBITRACIÓN [12]

Respecto a la velocidad de transmisión del bus, la norma ISO 11898 [1] especifica, para una red de máximo 30 nodos y una longitud de bus de 40 m, una velocidad de transmisión máxima de 1 Mbps. En el presente proyecto se han empleado 3 nodos, menos de 1 metro de longitud y una velocidad de 125 Kbps.

Table 1. Suggested Cable Length vs Signaling Rate

Bus Length (m)	Signaling Rate (Mbps)
40	1
100	0.5
200	0.25
500	0.10
1000	0.05

FIGURA III - 3 – RELACIONES MÁXIMAS ENTRE LONGITUD DEL BUS Y VELOCIDAD DE ENVÍO DE DATOS [1]

La **capa de enlace** define la trama, que es la forma en que se organizan los bits a la hora de transmitirlos por el canal.

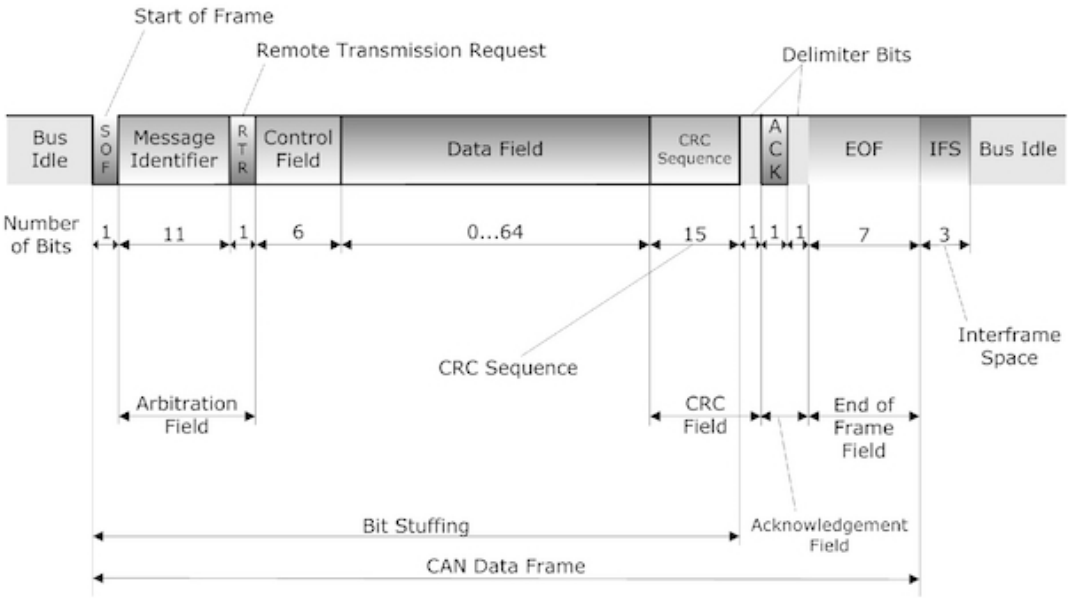

FIGURA III - 4 – CAPA DE ENLACE (TRAMA) DEL BUS CAN [13]

Los campos más relevantes son:

- **SOF (Start Of Frame):** bit con el que se anuncia la intención de transmitir, sirve para sincronizar los nodos
- **Identificador:** en la trama estándar, de 11 bits, se usa para el proceso de arbitración con el fin de evitar colisiones
- **Campo de control:** de los 6 bits de esta zona, 4 bits están reservados para lo que se conoce como DLC (código de longitud de datos), que indica cuántos datos van a ser enviados a continuación. El valor máximo de DLC es 0x08.
- **Campo de datos:** cada dato tiene un tamaño predeterminado de 1 byte. Puede haber en cada mensaje CAN un máximo de 64 bits, es decir, 8 mensajes de 1 byte.
- Los campos **CRC y ACK**: detección de errores
- **EOF (End Of Frame)**: 7 bits recesivos, indican fin de la transmisión.

Como se puede observar, en el *struct* definido en la librería “can.h” que se menciona en el Capítulo II, se definen un **can_id, can_dlc y data[]**, que son los componentes de la trama que necesitan ser modificados manualmente por el usuario en función de los requerimientos de cada mensaje.

Capítulo IV. Topología de la red

Capítulo IV. Topología de la red

IV.1.Diseño topológico de la red

La topología general de una red CAN tiene la siguiente estructura:

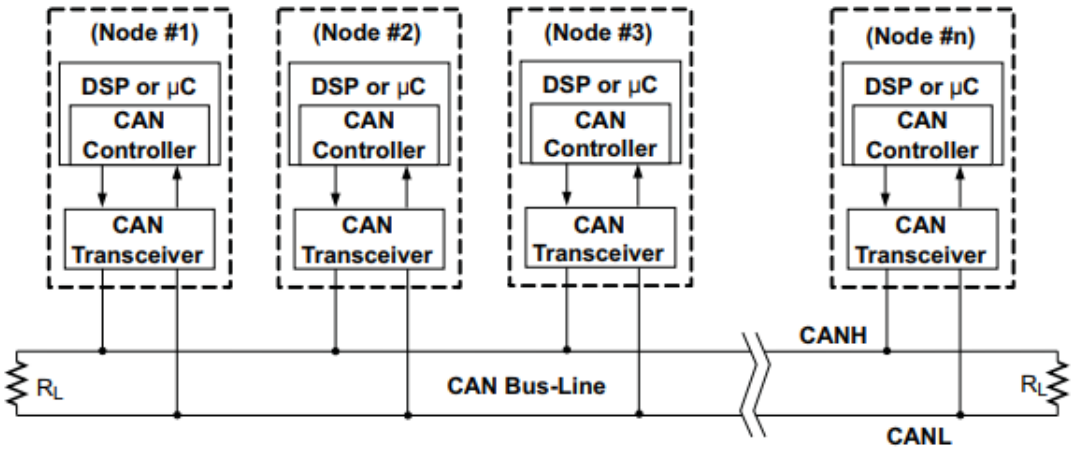

FIGURA IV - 1 – TOPOLOGÍA GENERALIZADA DE UNA RED CAN [1]

En este caso se ha diseñado la estructura de nodos según una división funcional, es decir, cada nodo se encarga de un bloque de funciones específicas. Estas funciones se han dividido en:

1. Medición constante y retransmisión de magnitudes físicas (revoluciones, velocidad y temperatura).
2. Gestión de luces de alumbrado y detección de errores.
3. Recepción y *display* de los datos provenientes de otros nodos, así como el posible envío de los mismos fuera del sistema vía radio/bluetooth.

Cabe destacar la característica mas interesante de este tipo de bus, y es su adaptabilidad. Si en el futuro fuera necesario introducir nuevos nodos (que incorporen nuevos bloques de función dentro del vehículo), o alterar el funcionamiento de los ya existentes (cambiando la prioridad de los mensajes o incluyendo nuevos), esto sería muy sencillo, debido a las características del bus:

- Dado que es un bus paralelo, la conexión de un nuevo nodo no requiere alterar las conexiones previas establecidas entre los anteriores, sino simplemente añadirlo a la red.

- Como el sistema se basa en prioridad de mensaje, el único requisito a cumplir por el nuevo nodo es tener en cuenta las prioridades de mensaje establecidas para los otros mensajes.
- Respecto al comportamiento del resto de nodos frente al nuevo elemento, sólo se debe realizar el correcto filtrado de mensajes de manera que únicamente los nodos interesados procesen los mensajes nuevos que les correspondan.

En conclusión, el diseño topológico se ha elegido en base a simular algunas necesidades principales del vehículo, pero se trata de un diseño flexible, que permite su modificación en el futuro sin demasiada complicación.

Respecto a la topología perteneciente a cada nodo por separado, se trata de la siguiente arquitectura:

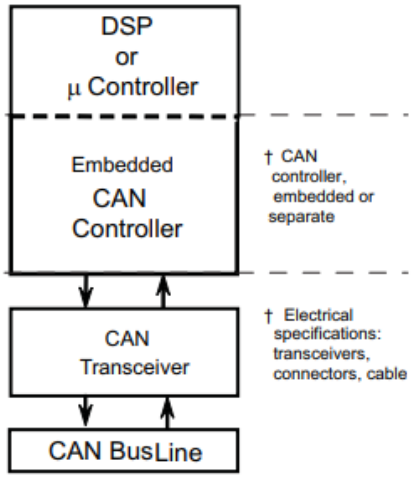

FIGURA IV - 2 – ESTRUCTURA DE UN NODO CAN [1]

En este proyecto, el controlador es el Arduino UNO, mientras que el controlador CAN y el *transceiver* están incluidos en el módulo MCP2515.

IV.2.Comunicación Arduino – MCP2515

Se comentará brevemente la estructura de comunicación entre el Arduino y el controlador CAN. Esto no es estrictamente topología de la red en sí, pero dado que se ha hablado de la arquitectura del nodo CAN y del diagrama de bloques del controlador, parece apropiado introducir algunos conceptos de comunicación que se encuentran en las funciones de la librería "mcp2515.h". De esta manera podemos explicar el funcionamiento de las funciones mencionadas en el Capítulo II, *mcp2515.reset(), mcp2515.sendMessage()* y *mcp2515.readMessage()*.

Tal y como se especifica en su datasheet [3], la estrategia de comunicación con el módulo CAN se basa en gestionar sus *buffers.* Tiene 3 *buffers* de transmisión de 14 bytes cada uno y 2 de recepción.

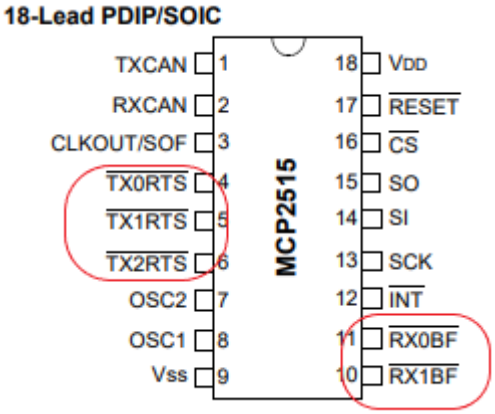

FIGURA IV - 3 – MODULO MCP2515 [3]

Dentro de cada registro TXnRTS se encuentran otra serie de registros. Los necesarios a conocer son:

- **TXBnCTRL:** registro de control.
- **TXBnSIDH, TXBnSIDL**: de los 11 bits que posee el identificador estándar, el TXBnSIDH contiene los 8 bits más significativos, y SIDL los 3 bits menos significativos.
- **TXBnDLC:** contiene el DLC del mensaje.
- **TXBnDm:** cada registro de los *m* totales es un byte de datos del mensaje.

El registro **TXBnCTRL** se usa para controlar las condiciones en las que se transmite el mensaje e indica el estado de la transmisión.

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

FIGURA IV - 4 – REGISTRO TXBNCTRL DEL MCP2515

TXBnCTRL.ABTF indica si el mensaje ha sido abortado o no, TXBnCTRL.MLOA si se ha ganado o perdido el proceso de arbitración, TXBnCTRL.TXERR si ha habido un error de transmisión, TXBnCTRL.TXREQ la petición de trasmitir un mensaje y TXBnCTRL.TXP (2 bits) indica, en caso de que hubiera varios mensajes en espera de ser enviados al mismo tiempo, cual debe ser enviado antes. Hay que tener cuidado con no confundir esta prioridad de mensajes con la prioridad en la arbitración del bus CAN, estamos hablando de dos cosas distintas.

Para inciar la comunicación, lo primero que se debe hacer es poner a uno el bit TXBnCTRL.TXREQ, lo que reseteará automáticamente los bits TXBnCTRL.ABTF, TXBnCTRL.MLOA y TXBnCTRL.TXERR. Esto no inicia la transmisión en sí misma, sólo indica al dispositivo que está listo para transmitir. La transmisión comenzará cuando el bus esté disponible.

En la figura IV-5 se observa el diagrama de flujo descrito arriba. No se va a entrar más en profundidad, dado que como se menciona anteriormente no entra dentro del alcance de este proyecto la gestión de *buffers*, solo se comenta brevemente su idea básica.

FIGURE 3-1: TRANSMIT MESSAGE FLOWCHART

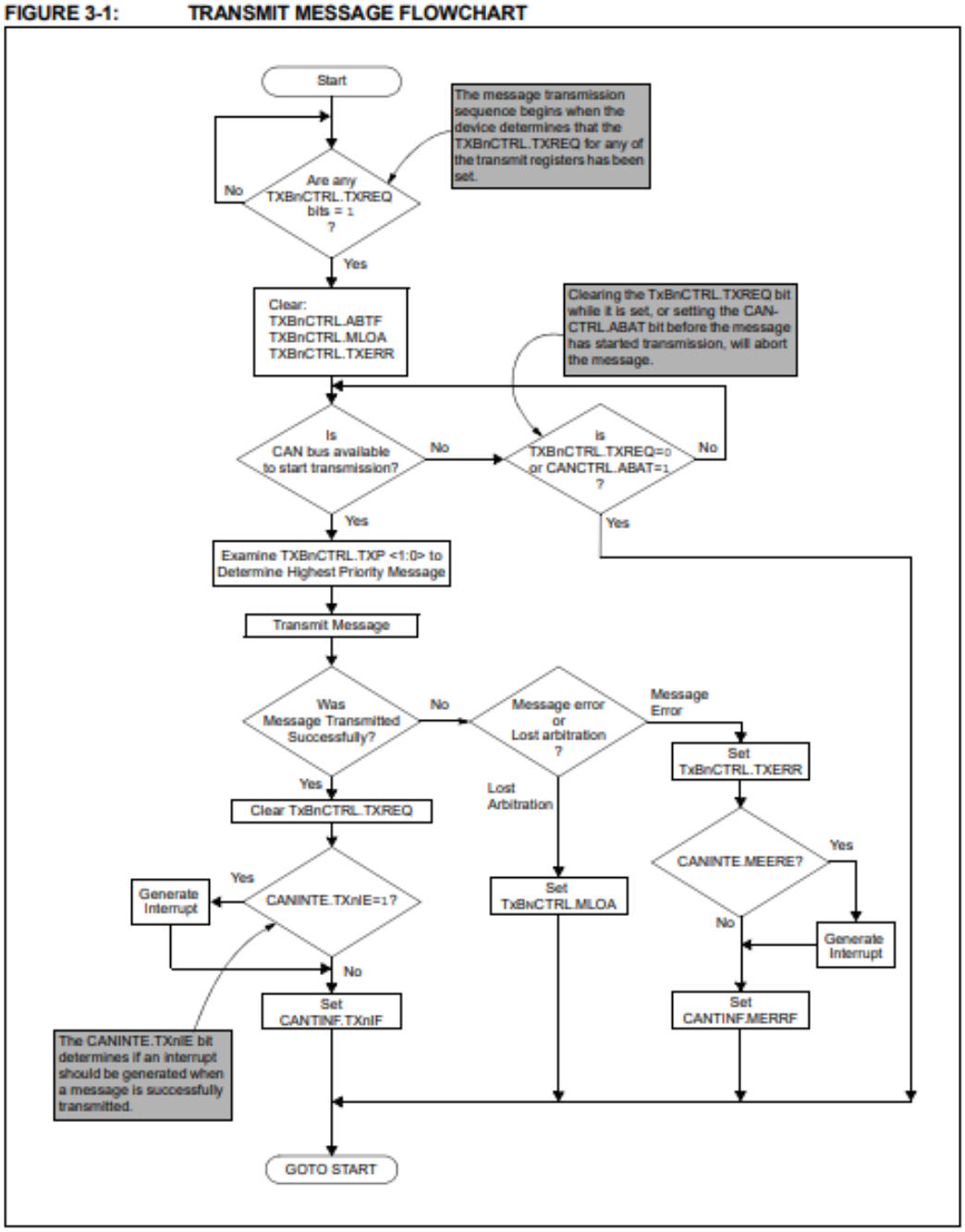

FIGURA IV - 5 – FLUJOGRAMA PARA ENVÍO DE MENSAJES CON EL MCP2515 [3]

Capítulo V. Nodos

Capítulo V. Nodos

V.1. Nodo de emisión constante

Este nodo es el único que manda mensajes de forma continua. Su función es la de medir regularmente las revoluciones en el árbol de levas, las revoluciones en las ruedas, calcular la velocidad (o comprobar la relación de marchas si se quisiera) y medir la temperatura, así como comunicar los datos periodicamente a través del bus.

La prioridad de estos mensajes es baja, dado que no son mensajes que representen urgencia, como pueden ser las alarmas por fallo de luz que se encuentran en los otros nodos. Se elige una prioridad baja de manera que si ocurre alguna urgencia en otra parte del vehículo ese otro mensaje; más relevante, pueda ganar el proceso de arbitración y emitir por el canal. El sistema completo se ha diseñado de manera que los mensajes de prioridad más alta sólo se envían en caso de emergencia (o en caso de reparación de dicha emergencia, como se explicará más adelante en este capítulo), de modo que en caso de interrumpir el flujo continuo de datos de este nodo, no lo hace por mucho tiempo, lo que es esencial, dado que no es permisible perder durante demasiado tiempo datos como la velocidad del coche, pues puede ser peligroso.

Como se puede observar en la figura V-2, se han elegido los DLCs de los mensajes en función de los valores máximos que queremos codificar en el coche. En un coche normal no se va a requerir codificar más de 255 km/h o 65535 rpm, aunque de todas formas se puede modificar la longitud de estos mensajes en caso de necesitarlo. Este nodo sólo tiene 3 entradas, que son los dos pines digitales de interrupción para calcular las revoluciones y un pin analógico para leer la temperatura. El código de este nodo consiste en una ejecución en bucle de las mismas 3 funciones, **calculateRPM(), calculateTemp() y startMessage().** A continuación se procederá a explicar su funcionamiento.

```
//Declaración de mensajes CAN
struct can_frame revoluciones1; //Revoluciones a la salida del motor
struct can_frame revoluciones2; //Revoluciones en la rueda
struct can_frame velocidad; //Velocidad real del coche; calculada a partir de las revoluciones en la rueda
struct can_frame temperatura;
MCP2515 mcp2515(10); //Pin del SS(Slave Select) del SPI
```

FIGURA V - 1 – DECLARACIÓN DE MENSAJES CAN

```
//Declaración de IDs y DLCs de los datos
velocidad.can_id = 0xA1;
velocidad.can_dlc = 1; //Se puede codificar hasta 255 km/h
revoluciones1.can_id = 0xA2;
revoluciones1.can_dlc = 2; //Se pueden codificar hasta 65535 rpm
revoluciones2.can_id = 0xA3;
revoluciones2.can_dlc = 2; //Se pueden codificar hasta 65535 rpm
temperatura.can_id = 0xA4;
temperatura.can_dlc = 1; //Se pueden codificar hasta 255°C
```

FIGURA V - 2 – DECLARACIÓN DE ID Y DLC DE LOS DATOS

```
pinMode(2, INPUT); //INT0
pinMode(3, INPUT); //INT1
pinMode(A1, INPUT); //Lectura de la NTC (temperatura)
```

FIGURA V - 3 – DECLARACIÓN DE PINES DEL ARDUINO UNO

```
void loop() {

  calculateRPM();
  calculateTemp();

  startMessage();

}
```

FIGURA V - 4 – FUNCIÓN "LOOP"

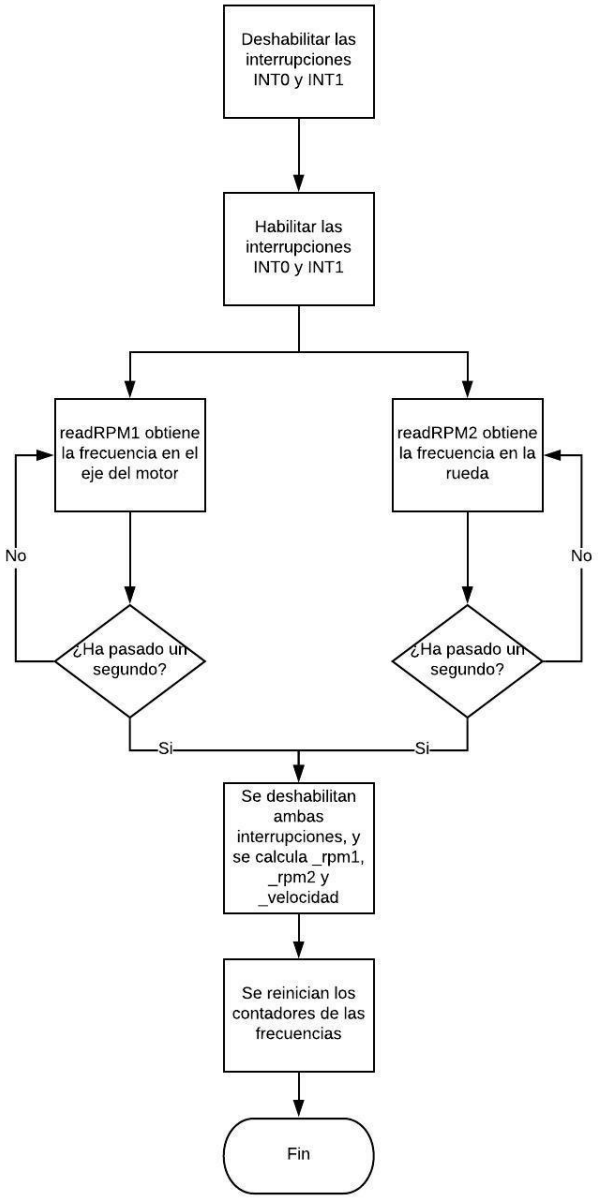

FIGURA V - 5 – DIAGRAMA DE FLUJO DE LA FUNCIÓN CALCULATERPM()

Lo primero que se hace es deshabilitar ambas interrupciones para, acto seguido, rehabilitarlas. De esta manera se asegura que si hubiera habido algun fallo en alguna deshabilitación anterior no haya un error de cuenta en esta iteración del proceso. La interrupción cero (INT0) dispara la función **readRPM1()** con cada flanco positivo de reloj del pin digital. Esta función cuenta los flancos de reloj durante un segundo, lo cual es una medida de la frecuencia. Lo anterior de aplica a la interrupción uno (INT1) y a **readRPM2().** Una vez ha terminado el segundo y se han obtenido ambas frecuencias, se vuelven a deshabilitar las interrupciones y se procede al cálculo de las revoluciones y la velocidad.

Dado que tenemos el dato de la frecuencia (vueltas por segundo), para obtener las revoluciones por minuto solo tenemos que multiplicar por 60. Obviamente hay una suposición implicita a este cálculo, y es que el comportamiento será lineal, o lo que es lo mismo, que las revoluciones se van a mantener constantes a lo largo del minuto. Entender esta suposición es necesario, dado que tiene un papel importante en la relación entre muestreo y precisión.

Se podría considerar que un periodo de un segundo entre medida y medida es demasiado tiempo, por lo que se debería aumentar la velocidad de las muestras disminuyendo el tiempo de muestreo. Esto es posible, pero tiene un inconveniente, y es que si reducimos, digamos que a la mitad (0.5 segundos) el tiempo de muestreo, la velocidad de aparición de los datos en pantalla se duplica, pero la precisión se reduce a la mitad, porque en vez de multiplicar por 60 ahora tendríamos que hacerlo por 120, y cualquier error, no linealidad o imperfección que haya podido haber durante el muestreo está siendo multiplicado por dos.

Un caso extremo de esto (y la única forma posible de estar 100% seguros de la medida) es muestrear durante el minuto completo. Tendríamos el perfil exacto de velocidad durante ese minuto, pero solo podríamos ver el dato una vez en un minuto, lo cual es inaceptablemente lento. Esto plantea una decisión de diseño, y es decidir que pesa más, si la velocidad de visualización de los datos o la precisión de los mismos. Este baremo debe ser considerado por el diseñador, aunque es fácil de alterar en el código. En este caso se consideró apropiado un periodo de muestreo de un segundo.

```
_rpm1 = cuentaRPM1/numInterrupciones1 * 60;//Pasa de rev/s(frecuencia) a rpm
_rpm2 = cuentaRPM2/numInterrupciones2 * 60;
_velocidad = _rpm2*PI*diametroRueda*60/1000; //Calculo de velocidad en km/h

cuentaRPM1 = 0; //Reinicia la cuenta
cuentaRPM2 = 0;
```

FIGURA V - 6 – CÁLCULO DE REVOLUCIONES Y VELOCIDAD EN LA FUNCIÓN CALCULATERPM()

Para el cálculo de las revoluciones también se tiene en cuenta el número de interrupciones en la rueda, dado que no tiene por qué haber un solo imán en toda la vuelta, de modo que:

$$\text{RPM} = \frac{Frecuencia\left(\frac{imanes}{segundo}\right) * 60\ (\frac{segundos}{minuto})}{Nº\ imanes\ (\frac{imanes}{vuelta})} = \text{n (vueltas/minuto)} \qquad \textbf{(1)}$$

El número de imanes por vuelta es un parámetro definido al principio del programa, de modo que sea fácilmente modificable para cualquier tipo de sensor/coche que usemos.

El cálculo de la velocidad se lleva a cabo teniendo en cuenta 3 factores. Las revoluciones que se usan son las de la rueda, pues tienen una relación directa con la velocidad del coche, no como el eje del motor. También se tiene en cuenta el diámetro de la rueda (otro parámetro que se define al principio del programa) y el intervalo de tiempo de muestreo, en nuestro caso aparece como 1000 ms.

$$\text{Velocidad} = \frac{Revoluciones\left(\frac{vueltas}{\text{min}}\right) * \pi * D_{rueda}(m) * 60(\frac{min}{h})}{1000(\frac{m}{km})} = \text{n (km/h)} \qquad \textbf{(2)}$$

```
#define diametroRueda (0.30)
#define numInterrupciones1 (7) //Define cuantas veces se va a ver interrumpido el haz en una sola vuelta
#define numInterrupciones2 (7) //Lo mismo pero en la rueda
```

FIGURA V - 7 – VARIABLES GLOBALES DEL NODO DE EMISIÓN CONSTANTE

Nótese que “vueltas” indica un número de eventos en un intervalo de tiempo. El tiempo tiene unidades, pero “vueltas” es adimensional. Una vez se han calculado todas las magnitudes, se reinician las cuentas de las rutinas de interrupción, y la función queda lista y en espera de la siguiente iteración.

```
void calculateTemp(){

  float VxD = analogRead(A1);
  float Vx = VxD*Vcc/1024;
  float Rx = (1300*Vcc/Vx) - 1300;
  _temp = (-1/0.032)*log(Rx/4606.2);
  /*
  Serial.print("Tª NTC: ");
  Serial.println(T);
  */
}
```

FIGURA V - 8 – FUNCIÓN CALCULATETEMP()

La temperatura se consigue a partir de la medida analógica del pin A1 del Arduino UNO. La función *analogRead()* devuelve un valor entre 0 y 1023, resultante de comparar el voltaje en ese pin con el de alimentación del Arduino, que en nuestro caso es de 5 V [2]. En la formula aparece como Vcc para poder alterar su valor fácilmente al principio del programa en caso de ser necesario. Dado que la NTC no da una salida de voltaje, sino de resistencia, es necesario un circuito acondicionador para convertir las variaciones de resistencia en variaciones de voltaje, para poder ser leído por el Arduino. Los detalles del circuito acondicionador de señal, así como de la ecuación para convertir la resistencia en temperatura, se explican en el siguiente capítulo.

En **startMessage()** se declaran tantos *String* como bytes de datos vayan a ser enviados, y se envían junto a una copia de los datos *_rpm1*, *_rpm2*, *_velocidad* y *_temp* a la función **toHex2byte()** o **toHex1byte()** dependiendo de si el mensaje a enviar tiene 1 o 2 byte de datos. No se entrará en profundidad en el funcionamiento de estas funciones (todo el código está adjunto en los anexos), son simplemente funciones que dan formato a las variables para poder ser enviadas por el bus CAN.

Una vez las funciones han dado formato a los *String*, estos se convierten a *Char* y luego a enteros en hexadecimal para, finalmente, ser asignados a los mensajes y enviados.

```
String revolucionesIData0;
String revolucionesIData1;
String revoluciones2Data0;
String revoluciones2Data1;
String velocidadData0;
String temperaturaData0;

int _rpmIAux = _rpmI; //Para no modificar los datos originales mientras se hace el cálculo
int _rpm2Aux = _rpm2;
int _velocidadAux = _velocidad;
int _tempAux = _temp;

toHex2byte(revolucionesIData0, revolucionesIData1, _rpmIAux);
toHex2byte(revoluciones2Data0, revoluciones2Data1, _rpm2Aux);
toHex1byte(velocidadData0, _velocidadAux);
toHex1byte(temperaturaData0, _tempAux);
```

FIGURA V - 9 – FUNCIÓN STARTMESSAGE() 1

```
revolucionesI.data[0] = number1;
revolucionesI.data[1] = number2;
velocidad.data[0] = number3;
temperatura.data[0] = number4;
revoluciones2.data[0] = number5;
revoluciones2.data[1] = number6;

sendMessage(1);
sendMessage(2);
sendMessage(3);
sendMessage(4);
```

FIGURA V - 10 – FUNCIÓN STARTMESSAGE() 2

```
void sendMessage(int a){ //Para poder enviar los mensajes con el delay(100), sin que ocurra error de segmentación
  switch(a){
    case 1: mcp2515.sendMessage(&revolucionesI); break;
    case 2: mcp2515.sendMessage(&revoluciones2); break;
    case 3: mcp2515.sendMessage(&velocidad); break;
    case 4: mcp2515.sendMessage(&temperatura); break;
  }
  delay(100);
}
```

FIGURA V - 11 – FUNCIÓN SENDMESSAGE()

V.2. Nodo de control de luces de alumbrado

Este nodo se encarga de leer el valor del switch rotativo que representa el selector de la palanca situada al lado del volante y encender o apagar las luces que hayan sido seleccionadas, así como hacer una revisión periódica del estado de las mismas, enviando un mensaje de prioridad alta por el canal en caso de haber encontrado un error.

```
struct can_frame ledError;
MCP2515 mcp2515(10);

#define inPin1 (2) //Luces de cruce + posicion
#define inPin2 (3) //Solo posición
#define inPin3 (4) //Luces largas + posición
#define inPin4 (5) //Luces antiniebla + posición
#define ledPinRed (6) //Luz cruce
#define ledPinBlue (7) //Luz larga
#define ledPinYellow (8) //Luz antiniebla
#define ledPinGreen (9) //Luz posición
```

FIGURA V - 12 – VARIABLES GLOBALES EN EL CÓDIGO DEL NODO DE GESTIÓN DE LUCES

```
ledError.can_id = 0x05; //Error importante
ledError.can_dlc = 4;
ledError.data[0] = 0x00; //Error cruce
ledError.data[1] = 0x00; //Error posición
ledError.data[2] = 0x00; //Error antiniebla
ledError.data[3] = 0x00; //Error larga
```

FIGURA V - 13 – DECLARACIÓN DEL MENSAJE "LEDERROR"

En este nodo se envía un único mensaje, que contiene 4 bytes de datos, donde cada byte representa el estado de cada uno de los tipos de luces que se han considerado para esta simulación.

La estructura del programa es la representada en la figura V-14. En primer lugar se lee la posición del selector, que es un switch rotativo, en el cual sólo se consideran las 4 primeras posiciones, dado que se han contemplado 4 combinaciones de luces (la posición del selector no enciende directamente la luz, sino que elige entre uno de los 4 programas disponibles):

- Luces de cruce + posición
- Solo posición

- Luces largas + posición
- Luces antiniebla + posición

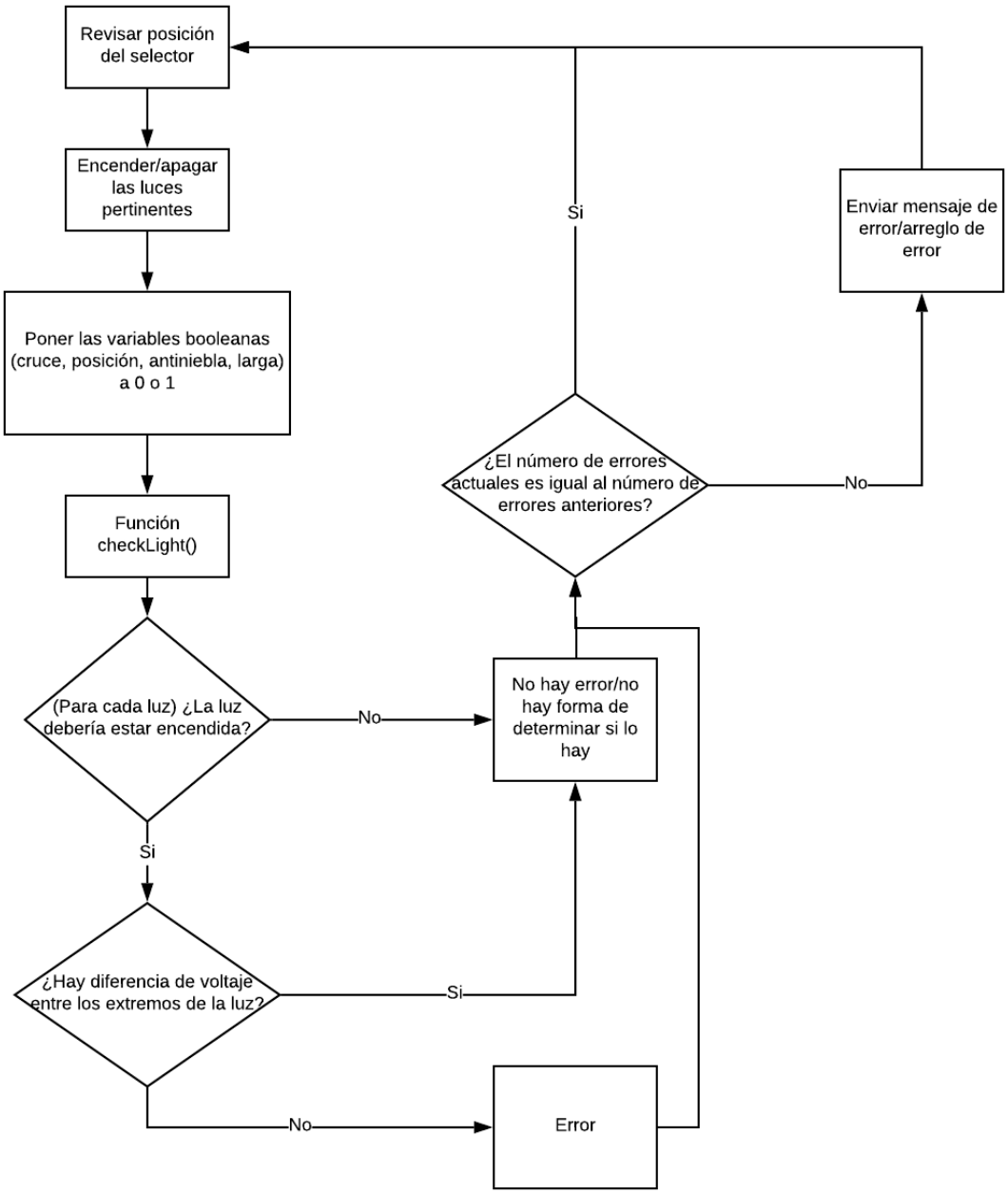

FIGURA V - 14 – FLUJOGRAMA DEL NODO DE GESTIÓN DE LUCES

Si se elige cualquier otra posición del switch, se apagarán todas las luces.

Tras comprobar el programa a utilizar, se encienden o apagan las luces pertinentes y se registra en una serie de variables booleanas si se ha ordenado o no encenderse a cada luz. Esto nos servirá posteriormente a la hora de comprobar si la luz funciona correctamente, pues solo podemos verificar su correcto funcionamiento si recordamos haber ordenado que se encienda.

```
//Luces de cruce + posicion
if(digitalRead(inPin1) == 1){
  digitalWrite(ledPinRed, HIGH);
  digitalWrite(ledPinGreen, HIGH);
  digitalWrite(ledPinYellow, LOW);
  digitalWrite(ledPinBlue, LOW);
  cruce = true;
  posicion = true;
  antiniebla = false;
  larga = false;
}
```

FIGURA V - 15 – PROCESO DE ENCENDER O APAGAR LUCES EN FUNCIÓN DE LA ENTRADA

Seguidamente se pasa a la función **checkLight()**, en la que se comprueba el estado del sistema. La primera condición para que una luz pueda ser verificada es que se le haya ordenado encenderse, no se puede verificar si una luz apagada es correcta o no (con el método diseñado para este proyecto).

Una vez sabemos si la luz debería estar encendida, se hace un lectura analógica del voltaje en ambos extremos de la resistencia previa al LED. Si estuviera encendida, una corriente estaría pasando a través de ella, de modo que debería haber una diferencia de potencial entre sus extremos. Si nuestras lecturas de voltaje coinciden, o bien son demasiado proximas, se asume que la luz es defectuosa.

Debido al número de medidas a realizar y la cantidad inferior de pines analógicos en el Arduino, se hace uso de un multiplexor 8 a 1 para ampliar la cantidad de entradas analógicas. Como se puede observar en la figura V-16, se selecciona la entrada del multiplexor mediante los pines A0, A1 y A2 (3 bits) para leer la salida del mismo por el pin A5.

```
//Comprobación luz de cruce
digitalWrite(A0, LOW); // 0
digitalWrite(A1, LOW);  // 0
digitalWrite(A2, LOW);  // 0
int highCruce = analogRead(A5);
digitalWrite(A0, LOW);   //  0
digitalWrite(A1, LOW);  //   0
digitalWrite(A2, HIGH);   // 1
int lowCruce = analogRead(A5);
if(highCruce > (lowCruce + 20) && cruce == true){ //MARGEN DE SEGURIDAD NECESARIO
  Serial.println("CRUCE: OK");
  errorVector[0] = 0;
}
else if(cruce == true){
  Serial.println("CRUCE: ERROR");
  errorVector[0] = 1;
}
```

FIGURA V - 16 – COMPROBACIÓN DE LA LUZ DE CRUCE

Una vez se ha determinado el número de errores actuales, se pasa a decidir si es necesario mandar un mensaje o no. Recordemos que este nodo no puede emitir continuamente por el canal, pues saturaría la transmisión de los demás, de manera que se decide sólo enviar el mensaje si ha habido información nueva, es decir, si el número actual de errores es diferente al número de errores de la anterior iteración.

Como se puede observar en la figura V-17, primero se revisa un vector donde se ha almacenado el estado de los errores, siendo 0 o 1:

- ledError.data[0]: error de la luz de cruce
- ledError.data[1]: error de la luz de posición
- ledError.data[2]: error de la luz antiniebla
- ledError.data[3]: error de la luz larga

Si el error actual es distinto del pasado se envía el mensaje y se actualiza el valor para la siguiente iteración.

```
  for(int i = 0; i < 4; i++){
    if(ledError.data[i] == 0x01){
      currentError++; //Desde que haya un solo error, "currentError" será mayor que cero
    }
  }

  if(currentError != pastError){
    sendMessage(); //Sólo hace falta mandar un mensaje si el dato ha cambiado, así no se satura el canal
  }
  pastError = currentError; //Se actualiza el valor del error para la siguiente interacción
  currentError = 0; //Se reinicia el error para que no se acumule en la siguiente interacción*/

}
```

FIGURA V - 17 – DECISIÓN DE ENVÍO O NO DE MENSAJES AL BUS

V.3. Nodo *Sniffer*

Este nodo no emite información por el canal. Se encarga de escuchar los mensajes de los demás nodos y guardar su contenido con el fin de ejercer de "base de datos", de manera que si se quisiera enseñar los datos por la centralita, o bien enviarlos por bluetooth, etc… se leyeran directamente las variables desde este nodo.

```
struct can_frame mensajeGeneral;
struct can_frame revoluciones1;
struct can_frame revoluciones2;
struct can_frame velocidad;
struct can_frame temperatura;
struct can_frame ledError;
MCP2515 mcp2515(10);
```

FIGURA V - 18 – DELARACIÓN DE MENSAJES EN EL NODO "SNIFFER"

Se definen los mismos mensajes que sabemos que los otros nodos estarán enviando, con los mismos nombres y DLCs, y sus byte de datos inicializados a cero.

Lo primero que se hace es leer el bus y almacenar el contenido del mensaje en un "mensajeGeneral", para luego comparar este mensaje a la lista de IDs asociada a los mensajes que conocemos de otros nodos para posteriormente volcar los datos del mensaje general en el mensaje apropiado que corresponda.

Velocidad	0xA1 ó 161	Prioridad baja
Revoluciones 1	0xA2 ó 162	Prioridad baja
Revoluciones 2	0xA3 ó 163	Prioridad baja
Temperatura	0xA4 ó 164	Prioridad baja
Luces	0x05 ó 5	Prioridad alta

```
if(mcp2515.readMessage(&mensajeGeneral)){
    String comparacion = String(mensajeGeneral.can_id);

    if(comparacion == "161"){//VELOCIDAD
      velocidad.can_id = mensajeGeneral.can_id;
      velocidad.data[0] = mensajeGeneral.data[0];
    }
    else if(comparacion == "162"){//REVOLUCIONES1
      revoluciones1.can_id = mensajeGeneral.can_id;
      revoluciones1.data[0] = mensajeGeneral.data[0];
      revoluciones1.data[1] = mensajeGeneral.data[1];
    }
    else if(comparacion == "163"){//REVOLUCIONES2
      revoluciones2.can_id = mensajeGeneral.can_id;
      revoluciones2.data[0] = mensajeGeneral.data[0];
      revoluciones2.data[1] = mensajeGeneral.data[1];
    }
    else if(comparacion == "164"){//TEMPERATURA
      temperatura.can_id = mensajeGeneral.can_id;
      temperatura.data[0] = mensajeGeneral.data[0];
      temperatura.data[1] = mensajeGeneral.data[1];
    }
```

FIGURA V - 19 – PROCESO DE COMPARACIÓN DE ID'S

Capítulo VI. Sensores y actuadores

Capítulo VI. Sensores y actuadores

VI.1.Sensores/actuadores de simulación

Los sensores y actuadores que han sido simulados han sido los del nodo de gestión de luces, debido a la dificultad e innecesariedad de implementar faros de automóvil.

Para simular el selector de luces se ha utilizado un switch rotativo del cual se utilizan las 4 primeras posiciones para elegir una de las 4 combinaciones posibles de luces.

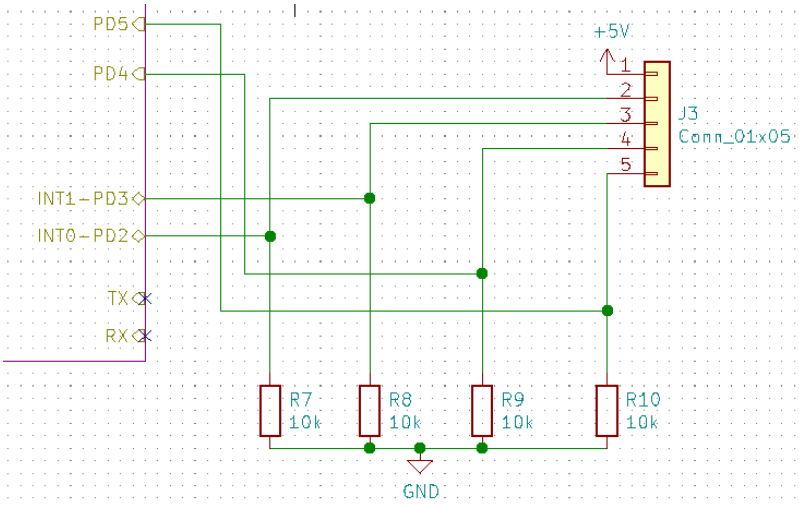

FIGURA VI - 1 – ESQUEMÁTICO DEL NODO DE GESTIÓN DE LUCES (1ª PARTE)

Se alimenta el switch por un extremo con los 5 voltios del Arduino, y este conmuta su terminal “maestro” con los pines 2, 3, 4 o 5 del conector J3. Estos está conectados a los pines 2, 3, 4 y 5 del Arduino, con el uso de resistencias de *pull-down*.

En función de estas señales se enciende la combinación de luces pertinente, que son simuladas a través de LEDs con resistencias en serie para evitar su sobrecarga.

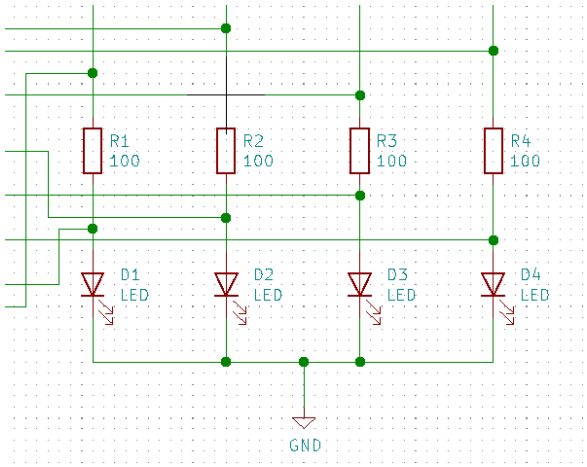

FIGURA VI - 2 – ESQUEMÁTICO DEL NODO DE GESTIÓN DE LUCES (2ª PARTE)

VI.2.Sensores/actuadores reales

Los sensores reales que se han utilizado pertenecen al nodo emisor. Se han utilizado los siguientes sensores:

- Sensor de efecto Hall para medir revoluciones en la rueda: **OEM 030 907 601**
- Sensor de efecto Hall para medir revoluciones en la caja de cambios: **OEM 357 919 149**
- Sensor NTC: **OEM 025 906 041**

Se muestra más información sobre los sensores utilizados en el anexo XI.1.

VI.2.1. Sensor de revoluciones 1 (a la salida del motor)

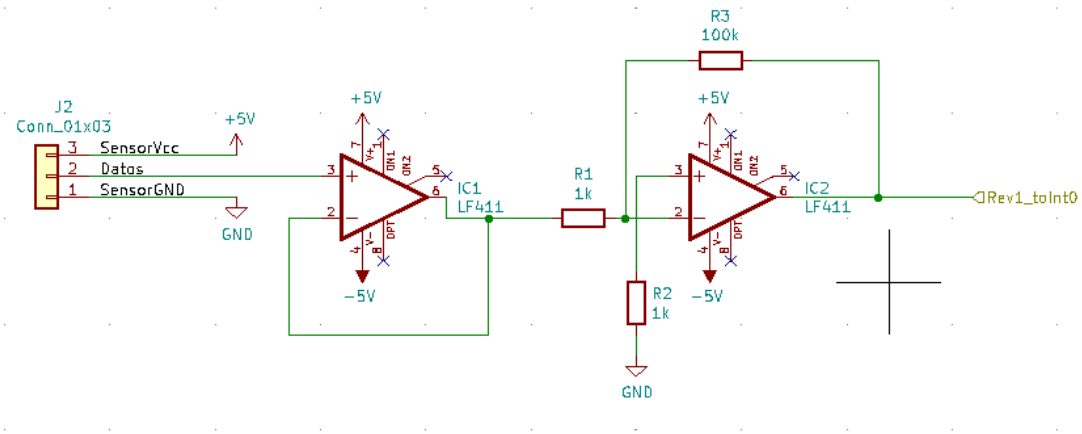

FIGURA VI - 3 ESQUEMÁTICO DEL ACONDICIONADOR DE SEÑAL PARA EL SENSOR DE REVOLUCIONES 1

El sensor de efecto Hall va conectado al cigüeñal o a la caja de cambios. Por si solo, genera un pico de 20 mV (lo que se observa en el osciloscopio, aunque sin demasiada precisión, dado que es una señal muy inestable) de tensión inversa cada vez que el giro pasa por la zona en la que se encuentra el imán. Para ser posible leerlo con el Arduino ha de superar la barrera que se considera “LOW” en digital, que es (para una alimentación de 5V) mayor de 3V.

El acondicionador de señal consta de dos partes:

1. **Seguidor de tensión** [9] [14]: se utiliza como etapa de entrada a la amplificación, dado que presenta una impedancia de entrada muy alta.

2. **Amplificador en configuración inversora** [9] [14]: dado que el sensor genera unos 20 mV de tensión inversa, debemos invertir la señal a la vez que la amplificamos. La ganancia de este amplificador viene dada por la fórmula 3:

$$A = -\frac{R3}{R1} = -\frac{100k}{1k} = -100. \quad \textbf{(3)}$$

Idealmente al aplicar esta ganancia se debería obtener una señal de unos 2 V en directa, pero sin embargo satura a 5V (que es la tensión de alimentación de los integrados), por lo que la estimación original de los 20 mV debe ser erronea.

Aun así esto es irrelevante, dado que como vamos a leer las vueltas del sensor usando una rutina de interrupción, lo único que nos importa es tener pulsos mayores de 3V (para poder ser leídos por el Arduino) y estables.

VI.2.1.1. Comprobación de las lecturas

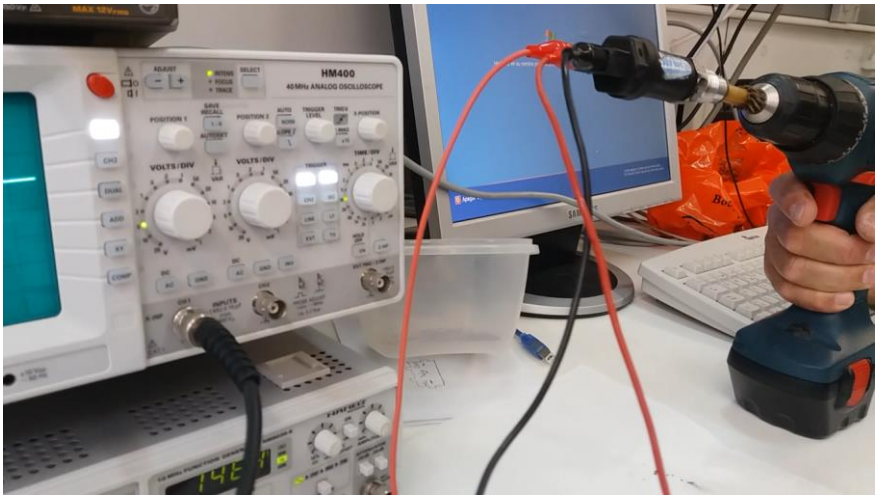

FIGURA VI - 4 – COMPROBACIÓN EXPERIMENTAL DEL SENSOR DE REVOLUCIONES 1

La comprobación experimental de las lecturas se realizó acoplando el sensor al eje de un taladro, y comprobando la lectura de frecuencia del Arduino con la mostrada por un frecuencímetro de instrumentación.

1. **Lectura 1: <u>39.2 Hz</u> → <u>40 Hz</u>**

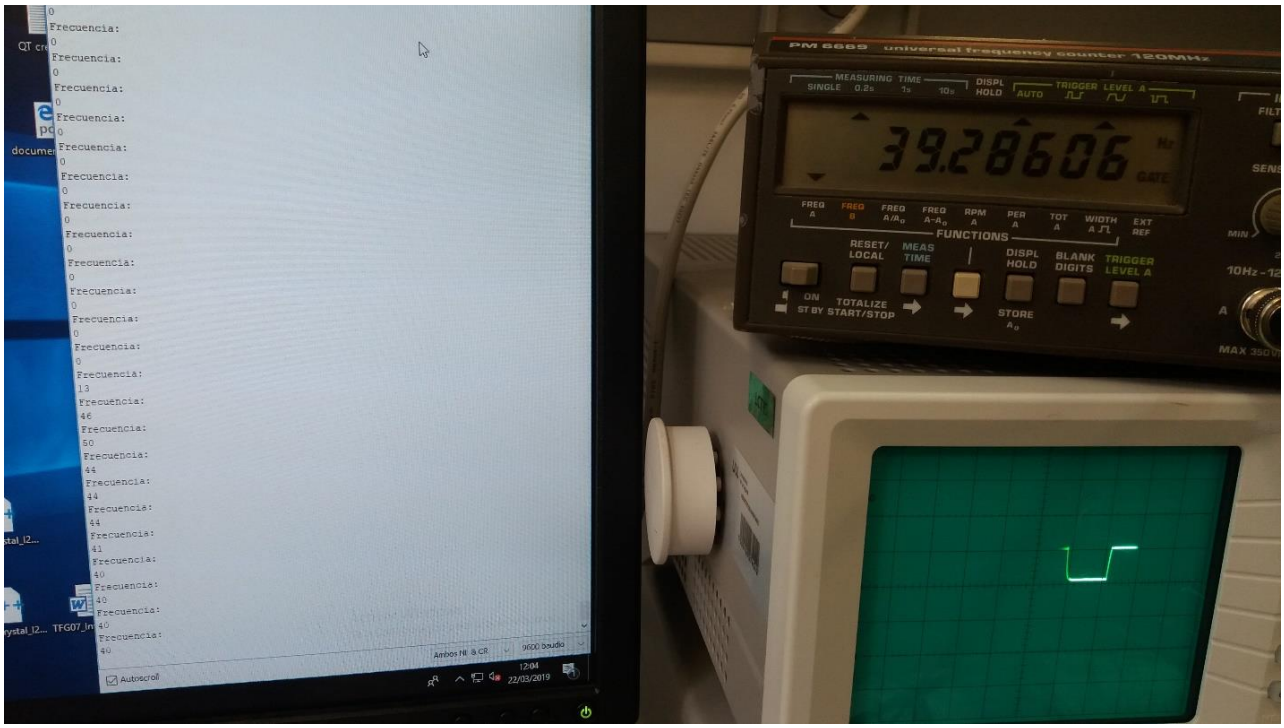

FIGURA VI - 5 – LECTURA 1

2. **Lectura 2: <u>83.2 Hz</u> → <u>84 Hz</u>**

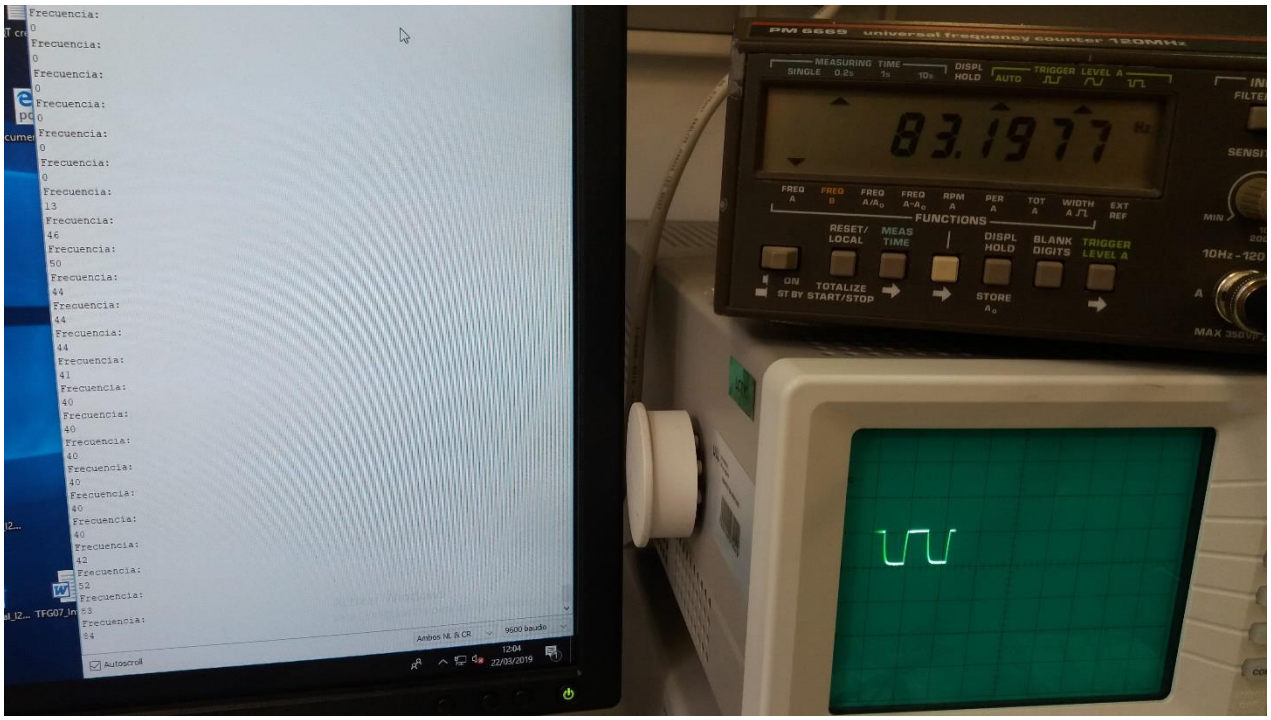

FIGURA VI - 6 – LECTURA 2

3. **Lectura 3: <u>56 Hz</u> → <u>57 Hz</u>**

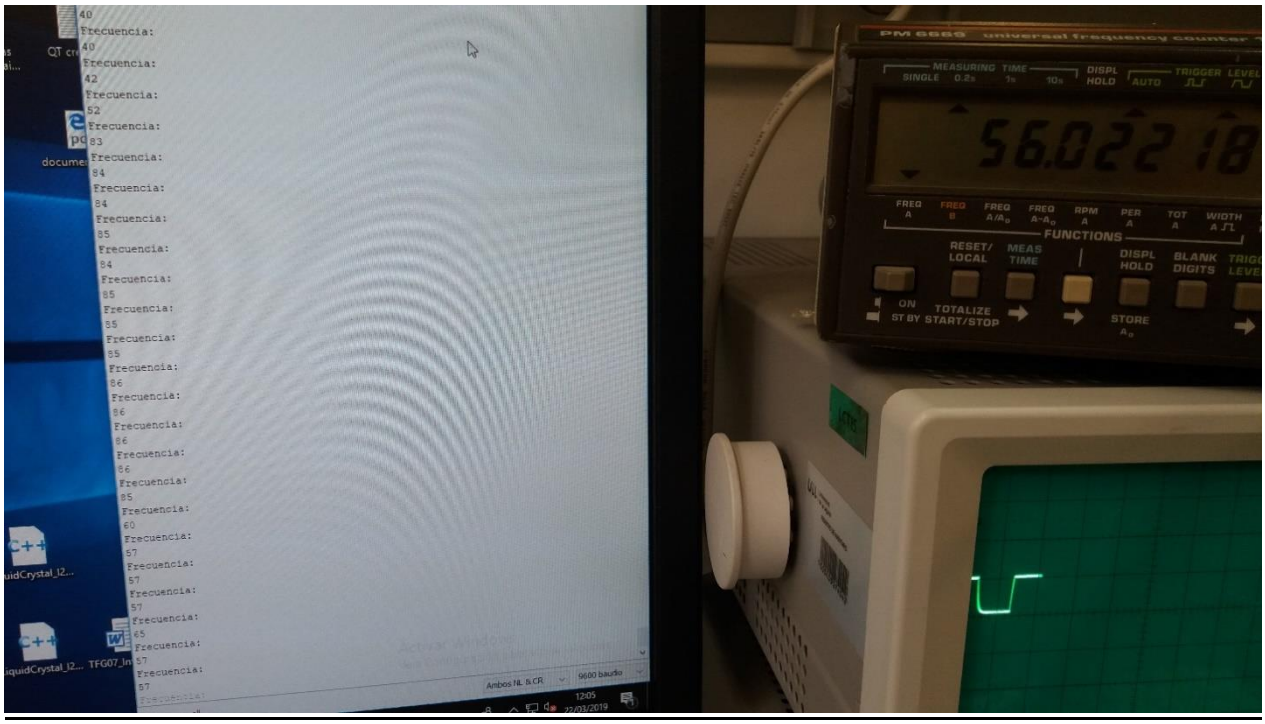

FIGURA VI - 7 – LECTURA 3

VI.2.2. Sensor de revoluciones 2 (en la rueda)

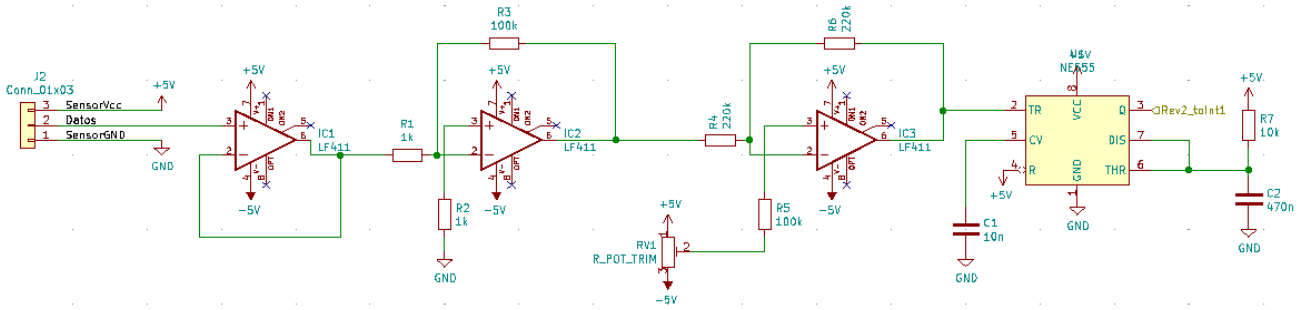

FIGURA VI - 8 - ESQUEMÁTICO DEL ACONDICIONADOR DE SEÑAL PARA EL SENSOR DE REVOLUCIONES 2

Este segundo sensor también es de efecto Hall. La principal diferencia con el primero es que no contiene el sensor Hall y el imán en el mismo dispositivo, sino que el imán es externo al sensor. Este se situa en la rueda del vehículo, mientras que el sensor se coloca en una posición fija alineada con la rueda. Este método representa una clara ventaja respecto al anterior, y es que al colocar los imanes de forma externa al sensor podemos elegir su número, pudiendo así colocar un mayor número y mejorar la resolución de la medida, pues en vez de tener datos una vez por cada vuelta completa de la rueda podemos acceder a fracciones, con lo que aumentamos la precisión.

La señal proveniente de este sensor tiene las mismas características que el anterior, con la salvedad de que en vez de 20 mV son aproximadamente 50 mV (también en inversa), y que tiene un ruido muy elevado, por lo que es necesario algún método para filtrarlo. Las dos primeras etapas del acondicionador son idénticas al anterior, pues se requiere de nuevo un seguidor como etapa de entrada seguido de un amplificador inversor. En total se dispone de:

1. **Seguidor de tensión.**
2. **Amplificador en configración inversora.**
3. **Comparador.** [9] [14]
4. **Temporizador NE555 en configuración monoestable.** [9] [14]

Las 2 primeras etapas ya han sido explicadas, de manera que se procederá con las dos últimas.

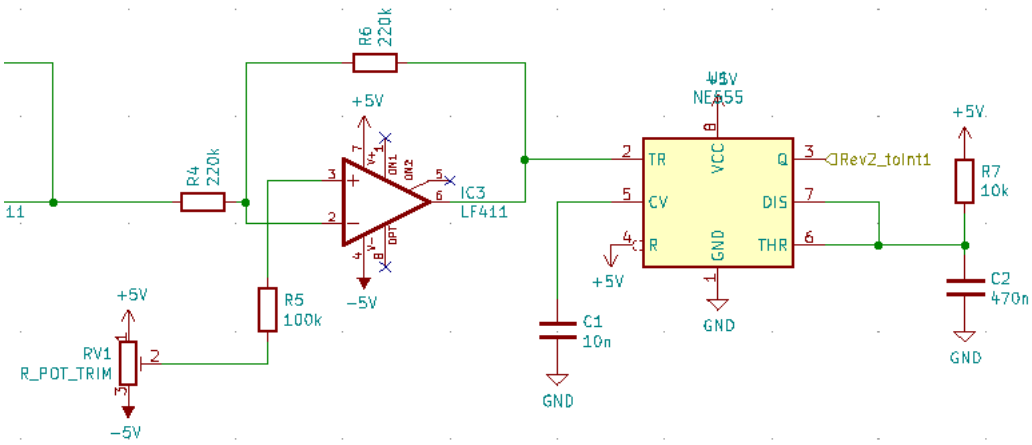

FIGURA VI - 9 ÚLTIMAS 2 ETAPAS DEL ESQUEMÁTICO DEL ACONDICIONADOR DE SEÑAL PARA EL SENSOR DE REVOLUCIONES 2

Empezaremos por el funcionamiento del NE555 [14]. En la configuración monoestable, la salida tiene 2 estados:

- Estado estable (nivel bajo): patilla 3 a 0V.
- Estado inestable (nivel alto): patilla 3 a tensión de alimentación, que en este caso son 5V.

El tiempo que se mantiene la señal de salida en nivel alto se ajusta con la resistencia R7 y el condensador C2 (figura VI-9). No es demasiado relevante este tiempo, siempre y cuando al Arduino le de tiempo a leer los flancos. Sigue la siguiente ecuación:

$$T = 1.1*R*C = 1.1*10^{3}*470^{-9} = \mathbf{5.17\ ms} \qquad \mathbf{(4)}$$

El paso de un estado a otro se controla con la patilla 2 (*trigger*). Esta patilla debe actuar como un pulsador, normalmente tiene un valor cercano a la tensión de alimentación, y cuando desciende de un cierto umbral se dispara el estado alto de la patilla de salida, como se muestra en la VI-10:

FIGURA VI - 10 – FUNCIONAMIENTO DEL NE555 EN CONFIGURACIÓN MONOESTABLE

Se utiliza la señal proveniente del sensor para disparar la salida, limitando el ruido proveniente del sensor. Con las 2 primeras etapas se amplifica la señal, pero también el ruido. Al poder ajustar el nivel de disparo, podemos elegir aquel nivel que haga disparar la señal pero no el ruido, de manera que el NE555 actue como filtro. La forma de ajustar este umbral es madiante el comparador situado antes del monoestable.

En la figura VI-11 se puede observar como la señal (azul) se encuentra por encima del nivel de disparo (naranja), la cual activa el 555 (amarillo) tras el primer pulso negativo. Tras el disparo, la señal azul oscila, pero es irrelevante al 555, por lo que se previene el error de contar flancos de más con el Arduino.

FIGURA VI - 11 – DISPARO DEL MONOESTABLE EN EL EXPERIMENTO

El circuito comparador funciona de manera que la señal proveniente del amplificador entra por la patilla inversora, y en la patilla no inversora se ajusta un nivel de continua. Esto provoca que la salida del comparador siga la siguiente ecuación:

$$V_o = V_3 - V_{\text{señal invertida}} \qquad \textbf{(5)}$$

Lo que quiere decir la expresión es que la señal amplificada proveniente del sensor estará montada sobre un nivel de continua e invertida. Este nivel de continua está por encima del nivel necesario para disparar el 555, de manera que sólo un flanco provocado por un imán en el sensor disparará el monoestable. Si se ajusta bien el nivel de V_3 se consigue hacer disparar el 555 con los pulsos pero no con el ruido. Este nivel de continua se ajusta con el potenciometro, que aparece en la figura VI-9.

En la figura VI-12 se puede observar como la señal proveniente del sensor tiene ruido, además de que no todos sus pulsos “buenos” tienen la misma altura, por lo que hay que ajustar el nivel para que amplifique todos los pulsos “buenos” de menor amplitud, pero no tanto como para que el 555 pueda ser disparado con el ruido.

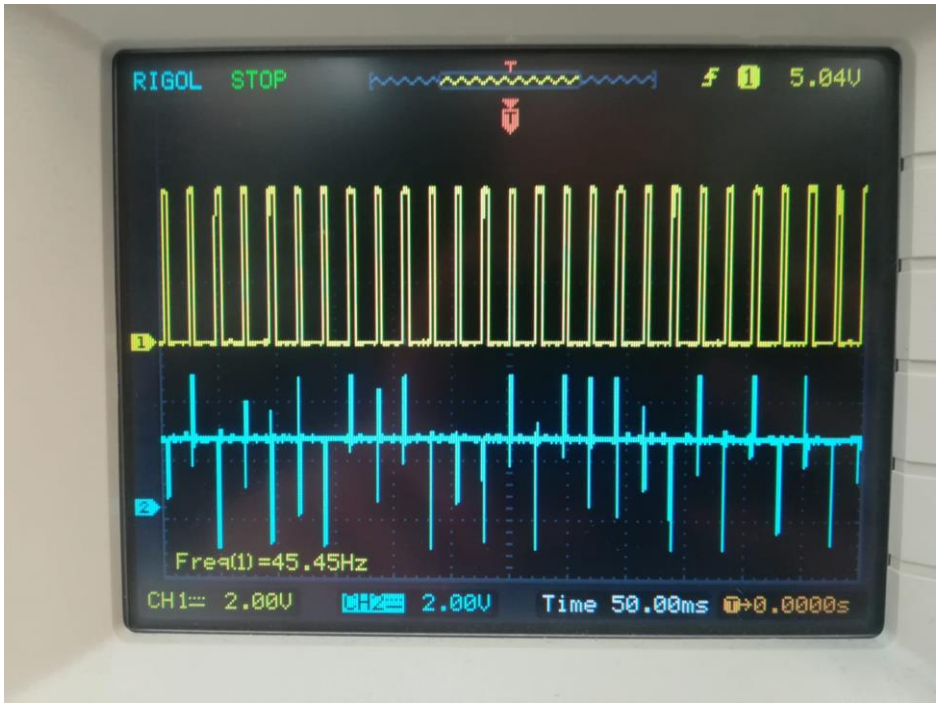

FIGURA VI - 12 – SEÑAL DE ENTRADA AL MONOESTABLE (AZUL) FRENTE A SEÑAL DE SALIDA (AMARILLO)

VI.2.2.1. Comprobación de las lecturas

En este caso la comprobación experimental de las lecturas se realiza acoplando una rueda con un imán al extremo de una fresadora a la que se le puede ajustar la frecuencia de giro, de modo que tenemos 3 fuentes de datos para comprobar: la frecuencia obtenida por nuestro circuito, la obtenida por un oscilosopio digital y la proporcionada por la propia fresadora.

1. **Lectura 1: <u>44 Hz → 44,64 Hz → 45 Hz</u>**

FIGURA VI - 13 – LECTURA 4

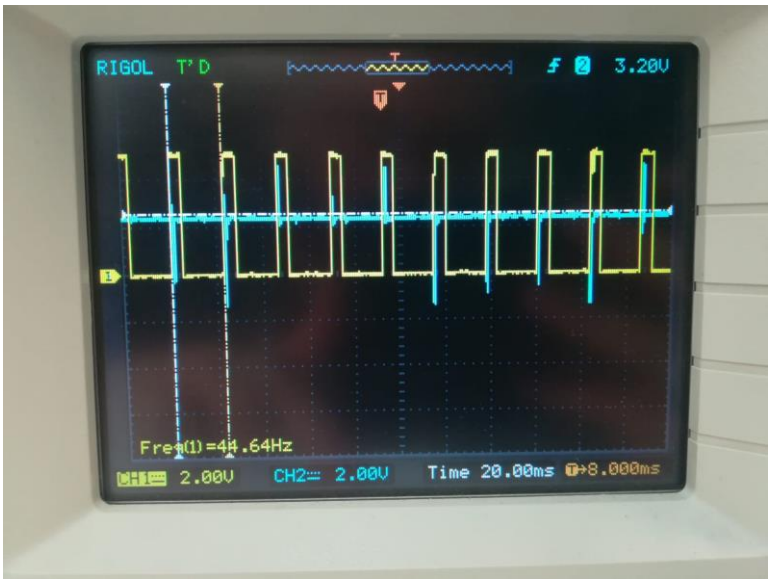

FIGURA VI - 14 – LECTURA 5

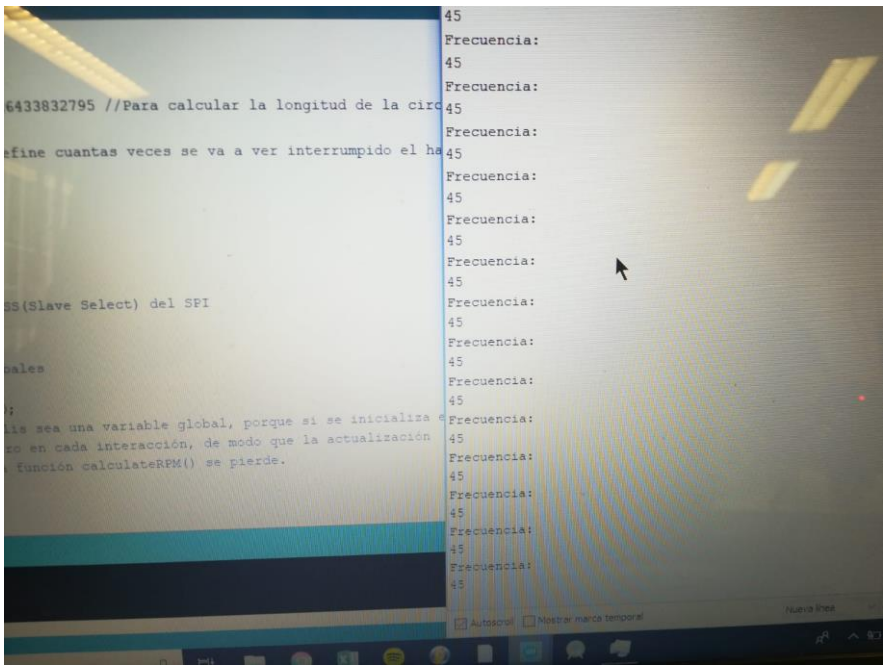

FIGURA VI - 15 – LECTURA 6

2. **Lectura 2: <u>61,2 Hz → 62,5 Hz → 62 Hz</u>**

FIGURA VI - 16 – LECTURA 7

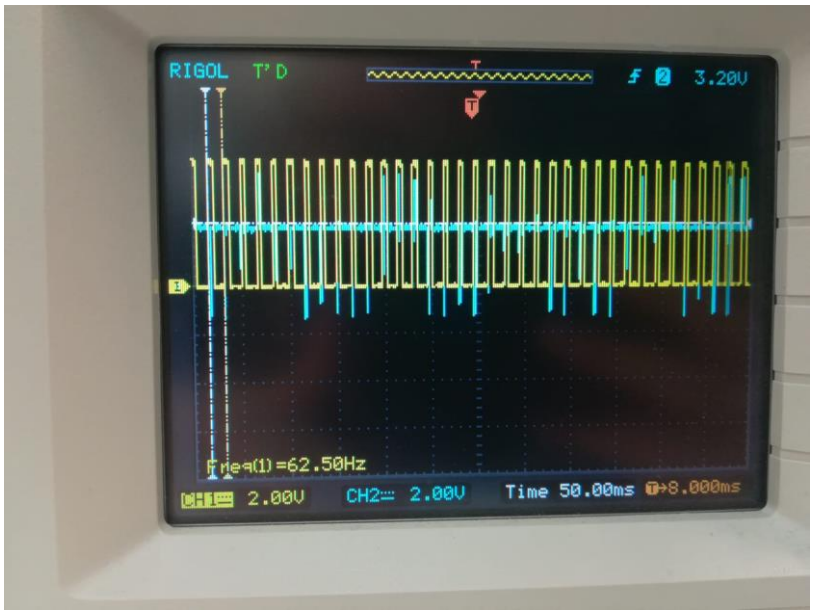

FIGURA VI - 17 – LECTURA 8

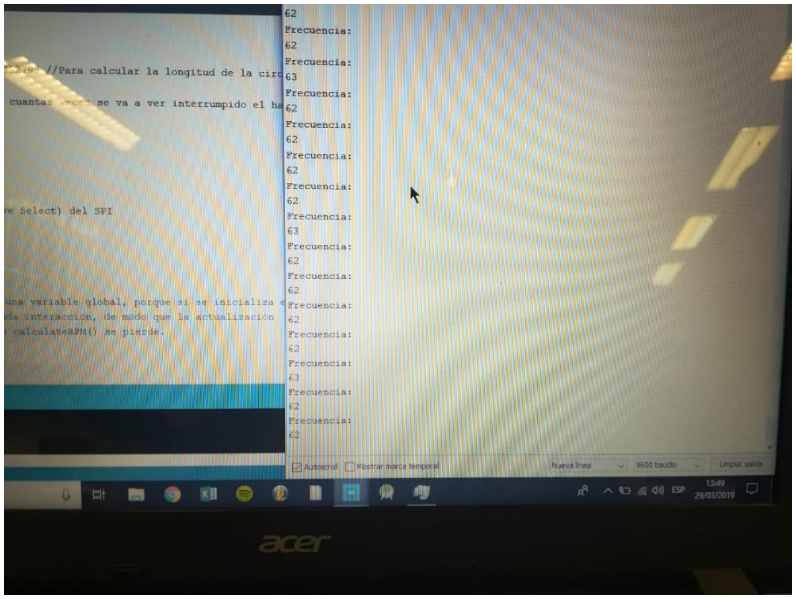

FIGURA VI - 18 – LECTURA 9

3. Lectura 2: <u>100 Hz → 100 Hz → 101 Hz</u>

FIGURA VI - 19 – LECTURA 10

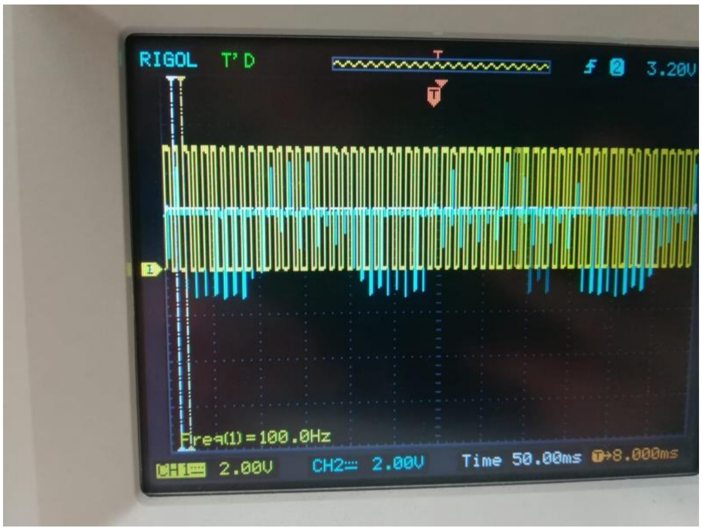

FIGURA VI - 20 – LECTURA 11

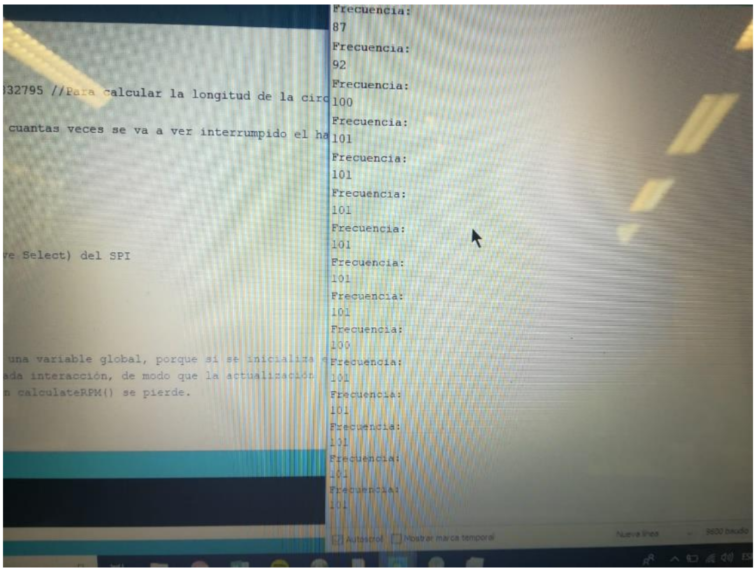

FIGURA VI - 21 – LECTURA 12

VI.2.3. Sensor de temperatura

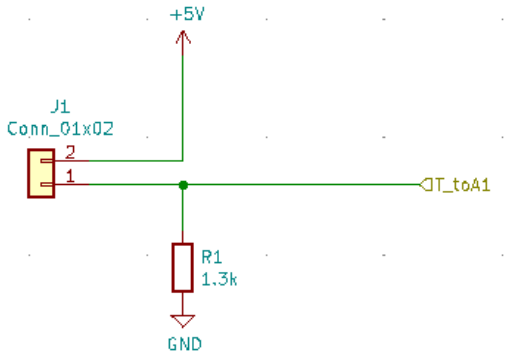

FIGURA VI - 22 – ESQUEMÁTICO DEL ACONDICIONADOR DE SEÑAL PARA LA NTC

El sensor de tempertura empleado es de tipo NTC. Como se explicó en la introducción, varía su resistencia en función de la temperatura [9], y ambas variables siguen una relación exponencial. Se trató de obtener dicha ecuación a través de su datasheet, pero no fue posible, de modo que se realizó una calibración para obtener la gráfica resistencia/temperatura.

Se colocó la NTC junto a un sensor tipo PT100 calibrado en el mismo vaso de precipitado, relleno de agua. Dicho vaso se colocó sobre un calentador/agitador magnético, que fue calentando progresivamente el agua, permitiendo tomar muestras cada incremento de medio ohmio de la PT100. Se medía:

- La temperatura obtenida con la PT100, a base de leer su resistencia y aplicar la ecuación 6:

$$T = \frac{R-100}{0.385} \qquad \textbf{(6)}$$

- La resistencia de la NTC

Conociendo ambos valores se generó una tabla, obtenida entre 22,78ºC y 96,1ºC, con la cual se obtuvo la siguiente gráfica (figura VI-23):

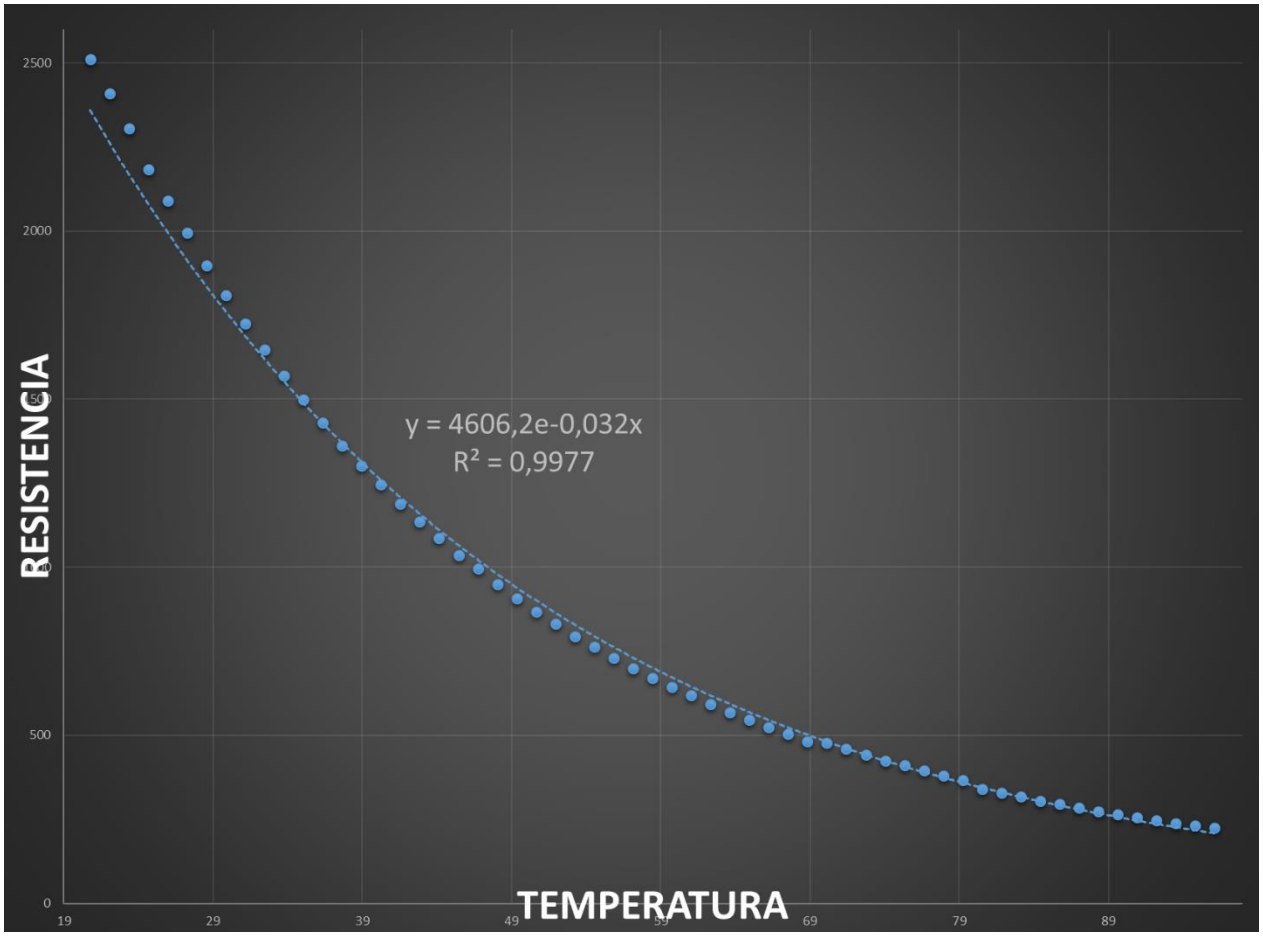

FIGURA VI - 23 – GRÁFICA DE RESISTENCIA FRENTE A TEMPERATURA DE LA NTC

Como se ha podido comprobar, la resistencia decae con la temperatura del fluido. Si tratamos de ajustar la nube de puntos a una ecuación de tipo exponencial, nos sale la ecuación mostrada en el gráfico, con un coeficiente de determinación de 0,9977, lo que implica que es muy buen ajuste. A continuación se explicará como hacer el factor de conversión entre la lectura en bits por parte del pin analógico y la temperatura final.

En la imagen se observa de derecha a izquierda la conversión desde la temperatura del fluido a resistencia por parte de la NTC, convertida a voltaje por medio del partidor de tensión mostrado al principio de este capítulo, y dicho voltaje convertido a un valor digital entre 0 y 1023 por medio del ADC del pin A1 del Arduino.

Si revertimos estas ecuaciones en orden inverso en el código del Arduino (esta fracción de código se enseña en el capítulo anterior), obtenemos la temperatura original a partir de la lectura digital del pin A1.

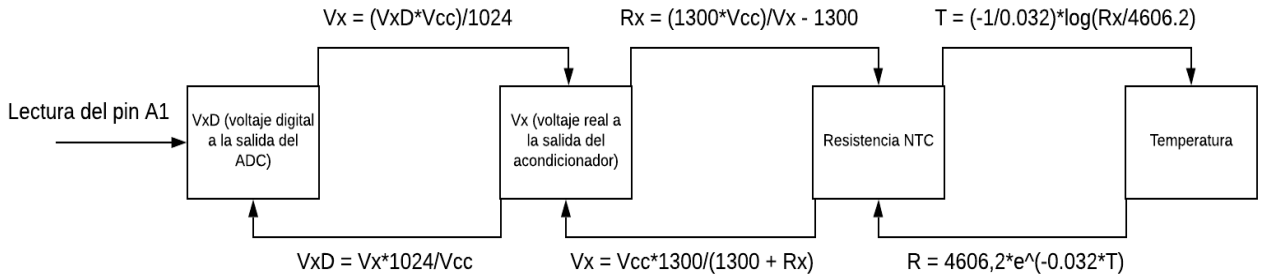

FIGURA VI - 24 – FUNCIONAMIENTO DEL ALGORITMO DE CÁLCULO DE TEMPERATURA

Capítulo VII. Presupuesto

Para el cálculo del presupuesto se tendrán en consideración los siguientes elementos, cuya suma es de **48,45€**

- Materiales presentes en la PCB del nodo de emisión constante: **20,4€**
- Materiales presentes en la PCB del nodo de gestión de luces de alumbrado: **8,9€**
- Arduino UNO usado para programar los Atmega328p y para ser utilizado como nodo *Sniffer:* **8,98€ (Amazon)**
- 3 módulos MCP2515: **3,39*3 = 10,17€ (eBay)**

Id	Nombre	Paquete	Cantida	Valor/modelo	Suministrador	Coste/unidad	Coste total
1	R6,R7,R8,R9,R10	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	5	10k	Yageo	0,09	0,45
2	R3,R1,R2,R4	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	4	100	Yageo	0,09	0,36
3	C4,C2,C3	C_Disc_D7.5mm_W5.0mm_P5.00mm	3	100n	Jb capacitors	0,06	0,18
4	C6,C5	C_Disc_D7.5mm_W5.0mm_P7.50mm	2	20p	Aliexpress	0,05	0,11
5	Y1	Crystal_HC50_Vertical	1	16M	Sparkfun	1,57	1,57
6	U2	DIP-16_W7.62mm	1	74LS151	eBay	1,85	1,85
7	D3,D1,D2,D4	LED	4	LED	Amazon	0,03	0,12
8	C1	CP_Radial_D8.0mm_P5.00mm	1	1u	Aliexpress	0,05	0,05
9	U3	DIP-28_W7.62mm	1	ATmega328P-PU	Aliexpress	1,48	1,48
10	D5	D_DO-34_SOD68_P12.70mm_Horizontal	1	DIODE	Aliexpress	0,01	0,01
11	J1	PhoenixContact_MSTBVA_2,5_2-G-5,08_1x02_P5.08mm_Vertical	1	Conn_01x02	eBay	1,56	1,56
12	J2	PinHeader_1x06_P2.54mm_Vertical	1	Conn_01x06	Aliexpress	0,02	0,02
13	J3	PinHeader_1x05_P2.54mm_Vertical	1	Conn_01x05	Aliexpress	0,02	0,02
14	R5	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm_Horizontal	1	1M	Yageo	0,09	0,09
15	SW1	SW_PUSH (4 pines)	1	Switch	eBay	0,30	0,30
16	U1	TO-220-3_Vertical	1	LM7805_TO220	eBay	0,72	0,72
							8,90

Id	Nombre	Paquete	Cantidad	Valor/modelo	Suministrador	Coste/unidad	Coste total
1	U4	DIP-8_W7.62mm	1	NE555	eBay	0,18	0,18
2	R1	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	1	1M	Yageo	0,09	0,09
3	R3,R4,R6,R7	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	4	1k	Yageo	0,09	0,36
4	R5,R8,R10	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	3	100k	Yageo	0,09	0,27
5	R9,R11	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	2	220k	Yageo	0,09	0,18
6	R13	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	1	1.3k	Yageo	0,09	0,09
7	R2,R12	R_Axial_DIN0207_L6.3mm_D2.5mm_P7.62mm	2	10k	Yageo	0,09	0,18
8	C14,C6,C5,C7,C8,C9,C10,C11,C12,C13,C15,C16,C17	C_Disc_D7.5mm_W5.0mm_P5.00mm	13	100n	Jb capacitors	0,06	0,78
9	J1	PhoenixContact_MSTBVA_2,5_3-G-5,08_1x03_P5.08mm_Vertical	1	Conn_01x03	eBay	1,56	1,56
10	C1,C2,C3,C4	CP_Radial_D8.0mm_P5.00mm	4	1u	Aliexpress	0,05	0,22
11	C18,C19	C_Disc_D7.5mm_W5.0mm_P7.50mm	2	20p	Aliexpress	0,05	0,11
12	C20	C_Disc_D7.5mm_W5.0mm_P5.00mm	1	10n	Aliexpress	0,05	0,05
13	C21	C_Disc_D7.5mm_W5.0mm_P5.00mm	1	470n	Aliexpress	0,05	0,05
14	D1	D_DO-34_SOD68_P12.70mm_Horizontal	1	DIODE	Aliexpress	0,01	0,01
15	IC1,IC2,IC3,IC4,IC5	LF411	5	LF411	eBay	2,10	10,50
16	J2	PinHeader_1x06_P2.54mm_Vertical	1	Conn_01x06	Aliexpress	0,02	0,02
17	J3,J4	PinHeader_1x03_P2.54mm_Vertical	2	Conn_01x03	Aliexpress	0,01	0,02
18	J5	PinHeader_1x02_P2.54mm_Vertical	1	Conn_01x02	Aliexpress	0,01	0,01
19	RV1	Potentiometer_Bourns_3296W_Vertical	1	R_POT_TRIM			0,00
20	SW1	Switch (4 pines)	1	Switch	eBay	0,30	0,30
21	U1	TO-220-3_Vertical	1	LM7805_TO220	eBay	0,72	0,72
22	U2	TO-220-3_Vertical	1	LM2990SX-5.0	eBay	1,61	1,61
23	U3	DIP-28_W7.62mm	1	ATmega328P-PU	Aliexpress	1,48	1,48
24	Y1	Crystal_HC50_Vertical	1	16M	Sparkfun	1,57	1,57
							20,37

Capítulo VIII. Aportaciones y conclusiones

El presente proyecto ha tenido como objetivo el diseño de un sistema de adquisición de datos y comunicación vía CAN basado en Arduino. Se ha conseguido:

- Crear una topología basada en la utilidad, con un total de 3 nodos CAN, cada uno gestionando una función fundamental del vehículo.
- Diseñar un sistema de comunicaciones basado en la prioridad de mensaje y enfocado en la optimización del canal de comunicación, siendo usado al máximo de su capacidad y sin perder información.
- Leer datos de 3 sensores reales mediante el diseño de 3 acondicionadores de señal.
- Simular un sistema de luces de alumbrado que se gestiona mediante un selector rotativo, que comprueba el estado de los faros y lo comunica en caso de ser necesario.
- Crear un repositorio de datos en el nodo *Sniffer*, que almacena todos los datos de los demas nodos para su visualización o futura exportación fuera del sistema.
- Diseñar 2 de los nodos en PCB, para mejorar su manejabilidad, reducir el cableado y minimizar el espacio innecesario ocupado por la *protoboard*.

Como posibilidades de mejora, se proponen las siguientes:

- La velocidad de envío de datos por el bus puede ser aumentada (dentro de lo que permite la norma ISO 11898), así como optimizar el uso de los bytes en cada mensaje, ya que el número máximo por mensaje es de 8 y, en el mejor de los casos, se usan 2 bytes, como en los mensajes de revoluciones.
- Implementar el sistema ya creado de gestión de luces de alumbrado para utilizar faros reales de automóvil, dado que la lógica del algoritmo es la misma, sólo es necesario añadir circuitos que amplifiquen las señales de control a señales de potencia (o emplear el uso de relés o similares).
- Aprovechar los datos almacenados en el nodo *Sniffer* para exportarlos fuera del sistema, ya sea mediante radio o bluetooth.

Capítulo IX. Bibliografía

[1] United Nations Standards Coordinating Committee, «Norma ISO 11898,» 2019. [En línea]. Available: http://www.ti.com/lit/an/slla270/slla270.pdf.

[2] Arduino, «Arduino CC,» 2019. [En línea]. Available: https://store.arduino.cc/arduino-uno-rev3.

[3] MICROCHIP, «Alldatasheet,» 2019. [En línea]. Available: http://www.alldatasheet.com/datasheet-pdf/pdf/166906/MICROCHIP/MCP2515.html.

[4] «Wikipedia,» 2019. [En línea]. Available: https://es.wikipedia.org/wiki/Bus_CAN#Historia_y_evoluci%C3%B3n_del_protocolo_CAN.

[5] «Circuit.io,» 2019. [En línea]. Available: https://www.circuito.io/blog/arduino-uno-pinout/.

[6] «Wikipedia,» 2019. [En línea]. Available: https://es.wikipedia.org/wiki/Serial_Peripheral_Interface.

[7] «Faranux,» 2019. [En línea]. Available: https://www.faranux.com/product/mcp2515-controller-bus-module-tja1050-receiver-spi-protocol-for-arduino/.

[8] P. A. Tipler y G. Mosca, FÍSICA, Barcelona [etc.] : Reverté, D.L. 2010., 2010.

[9] A. Malvino y D. Bates, PRINCIPIOS DE ELECTRONICA, McGraw-Hill Interamericana de España S.L., 2007.

[10] «GitHub,» 2019. [En línea]. Available: https://github.com/autowp/arduino-mcp2515.

[11] «Wikipedia,» 2019. [En línea]. Available: https://es.wikipedia.org/wiki/Bus_CAN#/media/File:Canbus_levels.svg.

[12] «Engineering White Papers,» 2019. [En línea]. Available: https://www.engineeringwhitepapers.com/test-measurement/characterizing-can-bus-arbitration/.

[13] «Copper Hill Technologies,» 2019. [En línea]. Available: http://www.copperhilltechnologies.com/can-bus-guide-message-frame-format/.

[14] O. B. González Hernández, S. E. Hernández Alonso y S. Rodríguez Pérez, Instrumentación Electrónica, 2013.

[15] «Wikipedia,» 2019. [En línea]. Available: https://en.wikipedia.org/wiki/Vehicle_bus.

Capítulo X. Glosario

CAN	**Controller Area Network**
PCB	**Printed Circuit Board**
NTC	**Negative Temperature Coefficient**
ABS	**Anti Brake System**
EBD	**Electronic Braking Distribution**
BAS	**Brake Assist System**
USB	**Universal Serial Bus**
INT0/INT1	**Interrupt 0/Interrupt 1**
SPI	**Serial Peripheral Interface**
CPU	**Central Processing Unit**
CS/SS	**Chip Select/Slave Select**
MISO/SO	**Master Input Slave Output/ Slave Output**
MOSI/SI	**Master Output Slave Input/Slave Input**
PTC	**Positive Temperature Coefficient**
ISO	**International Standarization Organization**
OSI	**Open System Interconnection**
ID	**Identifier**
DLC	**Data Length Code**
RPM	**Revoluciones Por Minuto**
LED	**Light Emitting Diode**
ADC	**Analog to Digital Conversion**

Capítulo XI. Anexos

XI.1. Sensores reales

El código OEM expresa la intercambiabilidad de las piezas, lo que significa que es irrelevante cuál fuera el sensor exacto utilizado, pues todos aquellos que comparten el número OEM tienen características equivalentes. La fuente de todos los sensores que se muestran a continuación es eBay.

XI.1.1. OEM 030 907 601

El sensor tiene un pin de alimentación de 5V, uno de tierra y uno de datos.

FIGURA XI - 1 – SENSOR 1

Compatibility

Please choose your vehicle's details for specific results.

Year -Select- | Make -Select- | Model -Select- | Trim -Select- | Engine -Select- | Go

[show all compatible vehicles]

This part is compatible with 34 vehicle(s).

Notes	Year	Make	Model	Trim	Engine
	2013	Audi	A8 Quattro	L W12 Sedan 4-Door	6.3L 6229CC 384Cu. In. W12 GAS DOHC Naturally Aspirated
	2012	Audi	A8 Quattro	L W12 Sedan 4-Door	6.3L 6229CC 384Cu. In. W12 GAS DOHC Naturally Aspirated
	2011	Volkswagen	Touareg	Comfortline Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2011	Volkswagen	Touareg	Highline Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2011	Volkswagen	Touareg	VR6 Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2010	Audi	Q7	Base Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2010	Audi	Q7	Premium Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2010	Volkswagen	Touareg	Comfortline Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2010	Volkswagen	Touareg	Highline Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2010	Volkswagen	Touareg	VR6 Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated
	2009	Audi	A3 Quattro	Base Hatchback 4-Door	3.2L 3189CC 195Cu. In. V6 GAS DOHC Naturally Aspirated
	2009	Audi	Q7	Base Sport Utility 4-Door	3.6L 3597CC 219Cu. In. V6 GAS DOHC Naturally Aspirated

FIGURA XI - 2 – COMPATIBILIDAD SENSOR 1

XI.1.2. OEM 357 919 149

El sensor tiene un pin de alimentación de 5V, uno de tierra y uno de datos.

FIGURA XI - 3 – SENSOR 2

	Audi	A3	1997	8L1 [1996-2003] Hatchback	1.8 T	Hatchback	1781ccm 150HP 110KW (Petrol)
	Audi	A3	1997	8L1 [1996-2003] Hatchback	1.8 T quattro	Hatchback	1781ccm 150HP 110KW (Petrol)
	Audi	A3	1997	8L1 [1996-2003] Hatchback	1.9 TDI	Hatchback	1896ccm 90HP 66KW (Diesel)
	Audi	A3	1997	8L1 [1996-2003] Hatchback	1.9 TDI	Hatchback	1896ccm 110HP 81KW (Diesel)
	Audi	A3	1996	8L1 [1996-2003] Hatchback	1.6	Hatchback	1595ccm 101HP 74KW (Petrol)
	Audi	A3	1996	8L1 [1996-2003] Hatchback	1.8	Hatchback	1781ccm 125HP 92KW (Petrol)
	Audi	A3	1996	8L1 [1996-2003] Hatchback	1.8 T	Hatchback	1781ccm 150HP 110KW (Petrol)
	Audi	A3	1996	8L1 [1996-2003] Hatchback	1.8 T quattro	Hatchback	1781ccm 150HP 110KW (Petrol)
	Audi	A3	1996	8L1 [1996-2003] Hatchback	1.9 TDI	Hatchback	1896ccm 90HP 66KW (Diesel)
	VW	Bora	2005	1J2 [1998-2005] Saloon	1.4 16V	Saloon	1390ccm 75HP 55KW (Petrol)
	VW	Bora	2005	1J2 [1998-2005] Saloon	1.6	Saloon	1595ccm 101HP 74KW (Petrol)

FIGURA XI - 4 – COMPATIBILIDAD SENSOR 2

XI.1.3. OEM 025 906 041

El sensor tiene 2 pines, los cuales representan ambos extremos de su resistencia interna.

FIGURA XI - 5 – SENSOR 3

bitte Artikelbeschreibung beachten	Audi	A6 Avant	1995/12-1997/12	4A, C4	1.8	1781 ccm, 92 KW, 125 PS
bitte Artikelbeschreibung beachten	Audi	A6 Avant	1995/12-1997/12	4A, C4	1.8 quattro	1781 ccm, 92 KW, 125 PS
bitte Artikelbeschreibung beachten	Audi	A6 Avant	1994/06-1997/12	4A, C4	1.9 TDI	1896 ccm, 66 KW, 90 PS
bitte Artikelbeschreibung beachten	Audi	A6 Avant	1994/06-1997/12	4A, C4	2.0	1984 ccm, 85 KW, 115 PS
bitte Artikelbeschreibung beachten	Audi	A6 Avant	1994/06-1997/12	4A, C4	2.0	1984 ccm, 74 KW, 100 PS
bitte Artikelbeschreibung beachten	Audi	Cabriolet	1993/01-1998/07	8G7, B4	2.0 E	1984 ccm, 85 KW, 115 PS
bitte Artikelbeschreibung beachten	Audi	Coupe	1989/05-1996/12	89, 8B	2.0	1984 ccm, 85 KW, 115 PS
bitte Artikelbeschreibung beachten	Seat	Cordoba	1994/06-2002/10	6K1, 6K2	1.4i	1390 ccm, 44 KW, 60 PS
bitte Artikelbeschreibung beachten	Seat	Cordoba	1996/09-2002/10	6K1, 6K2	1.4i 16V	1390 ccm, 74 KW, 101 PS
bitte Artikelbeschreibung beachten	Seat	Cordoba	1994/05-1999/06	6K1, 6K2	1.6i	1595 ccm, 55 KW, 75 PS
bitte Artikelbeschreibung beachten	Seat	Cordoba	1996/07-2002/10	6K1, 6K2	1.6i	1595 ccm, 74 KW, 101 PS
bitte Artikelbeschreibung beachten	Seat	Cordoba	1993/02-2002/10	6K1, 6K2	1.6i	1598 ccm, 55 KW, 75 PS
bitte Artikelbeschreibung beachten	Seat	Cordoba	1993/02-2002/09	6K1, 6K2	1.8i	1781 ccm, 66 KW, 90 PS

FIGURA XI - 6 – COMPATIBILIDAD SENSOR 3

XI.2. Diseño en PCB

XI.2.1. Esquemáticos

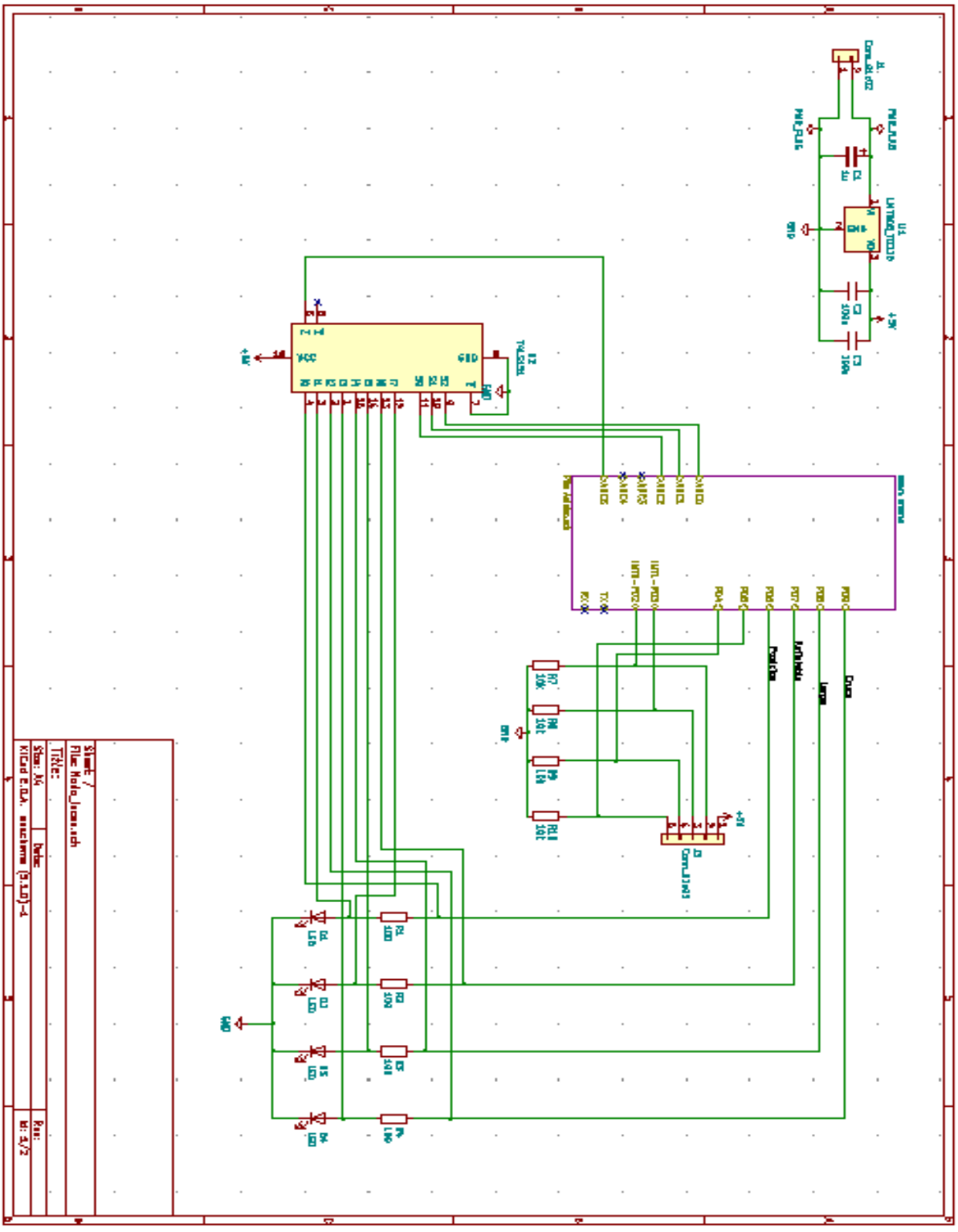

FIGURA XI - 7 – ESQUEMÁTICO DEL NODO DE GESTIÓN DE LUCES

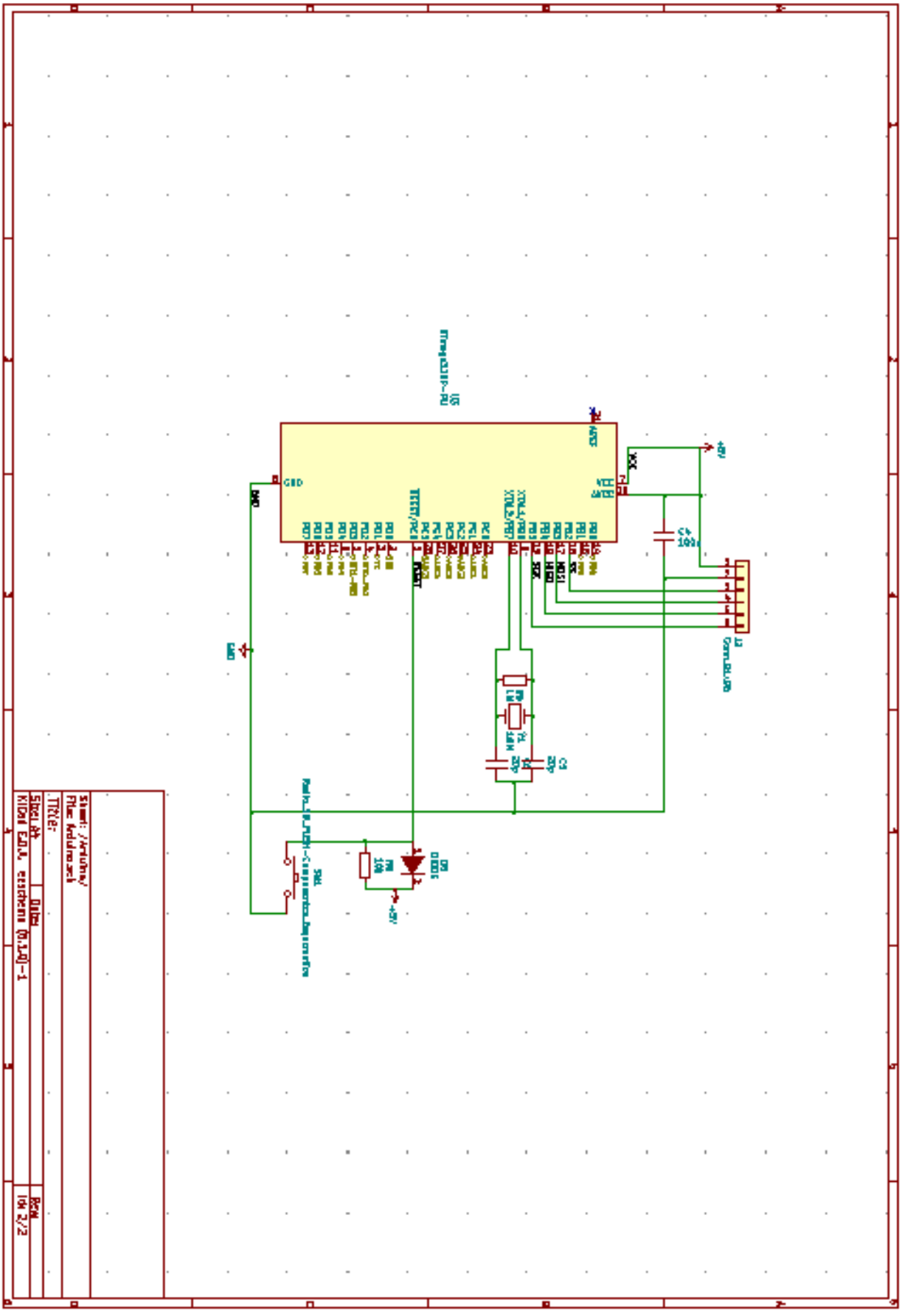

FIGURA XI - 8 – ESQUEMÁTICA DE LA SIMPLIFICACIÓN DEL ARDUINO UNO PARA IMPLEMENTACION EN PCB

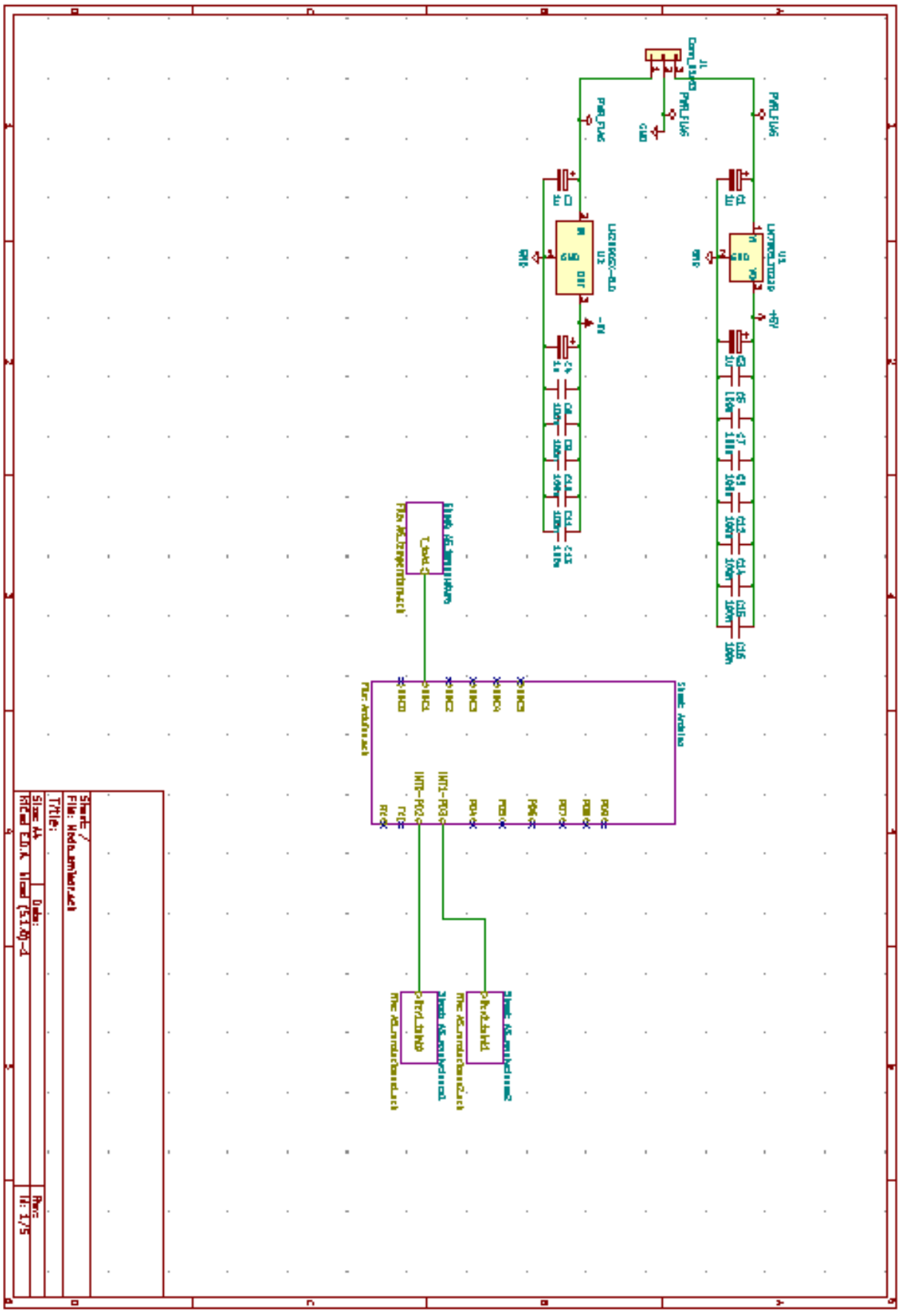

FIGURA XI - 9 – NODO DE EMISIÓN CONSTANTE

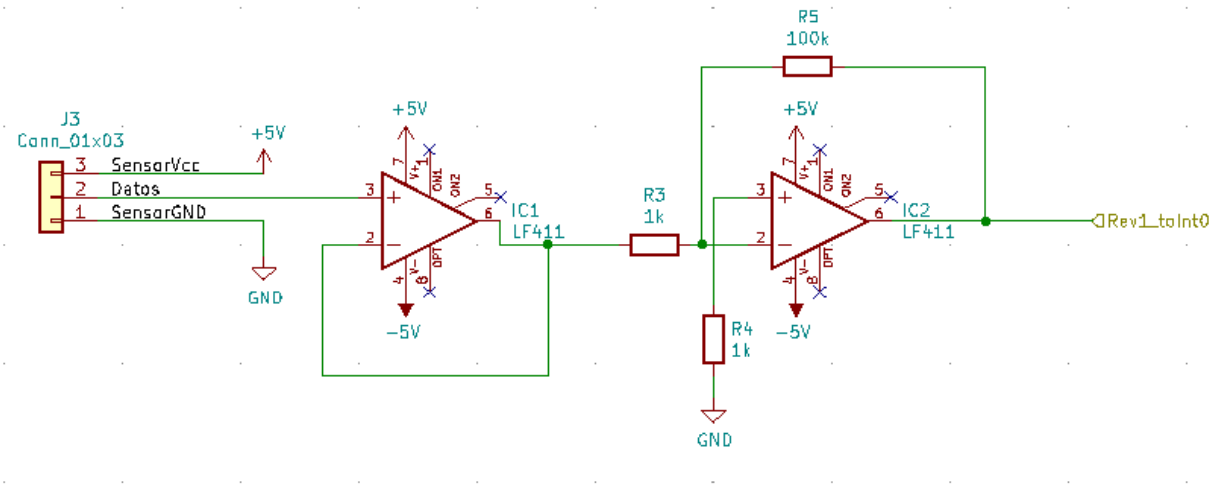

FIGURA XI - 10 – ACONDICIONADOR DE SEÑAL 1 DEL NODO DE EMISIÓN CONSTANTE

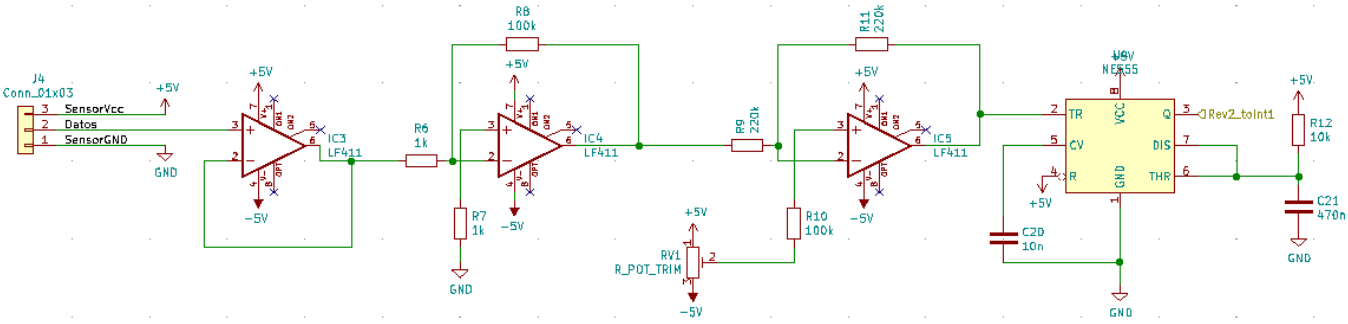

FIGURA XI - 11 - ACONDICIONADOR DE SEÑAL 2 DEL NODO DE EMISIÓN CONSTANTE

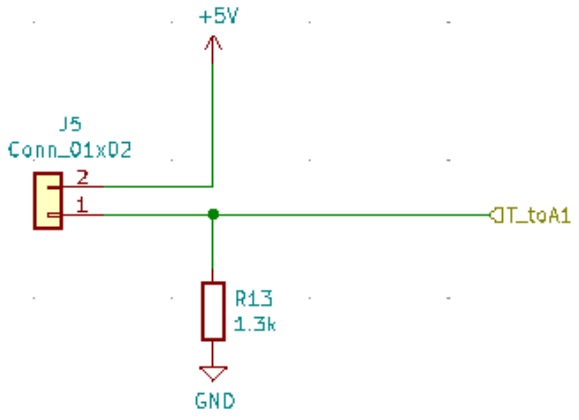

FIGURA XI - 12 - ACONDICIONADOR DE SEÑAL 2 DEL NODO DE EMISIÓN CONSTANTE

XI.2.2. Layout

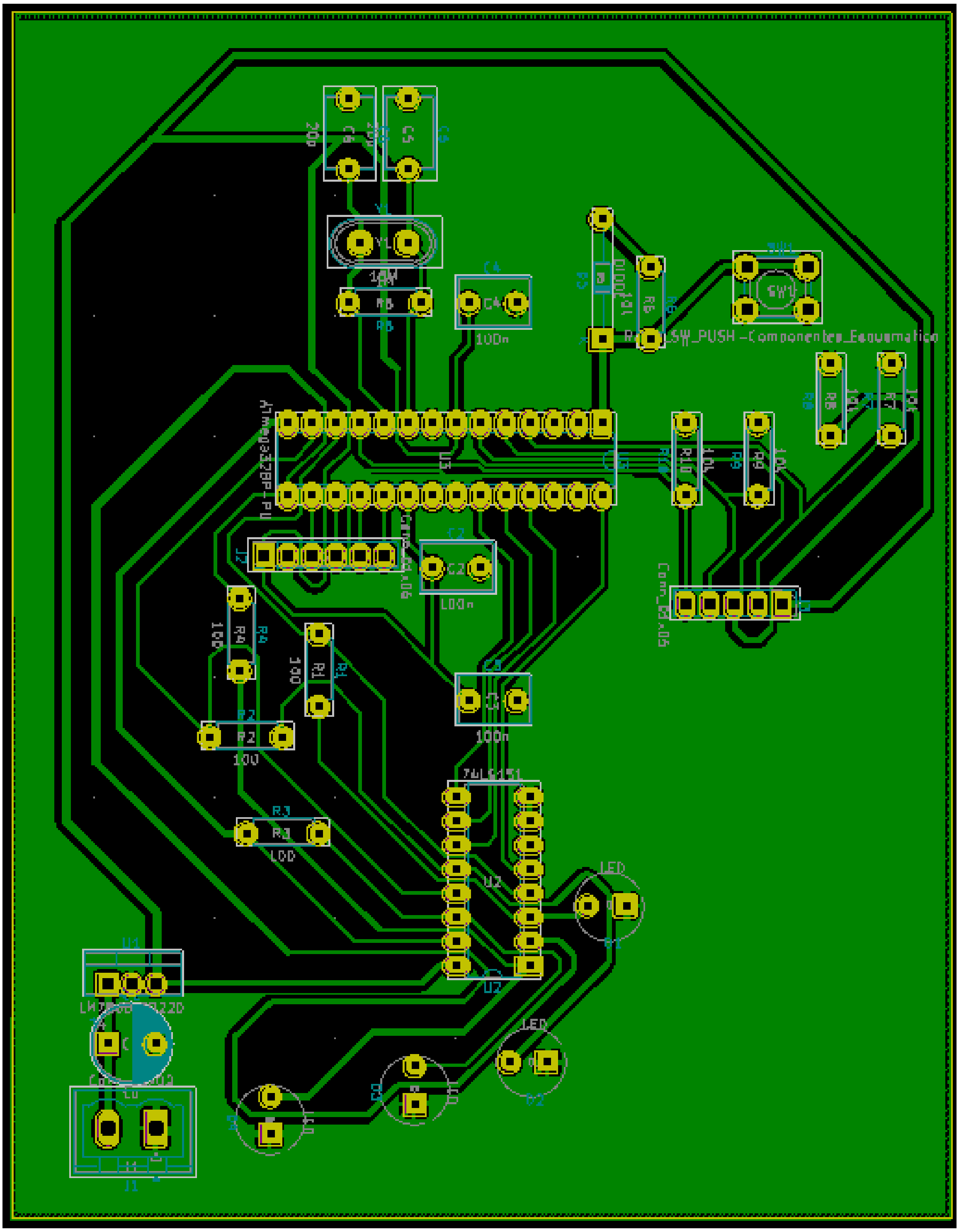

FIGURA XI - 13 – CARA BOTTOM DEL NODO DE GESTIÓN DE LUCES

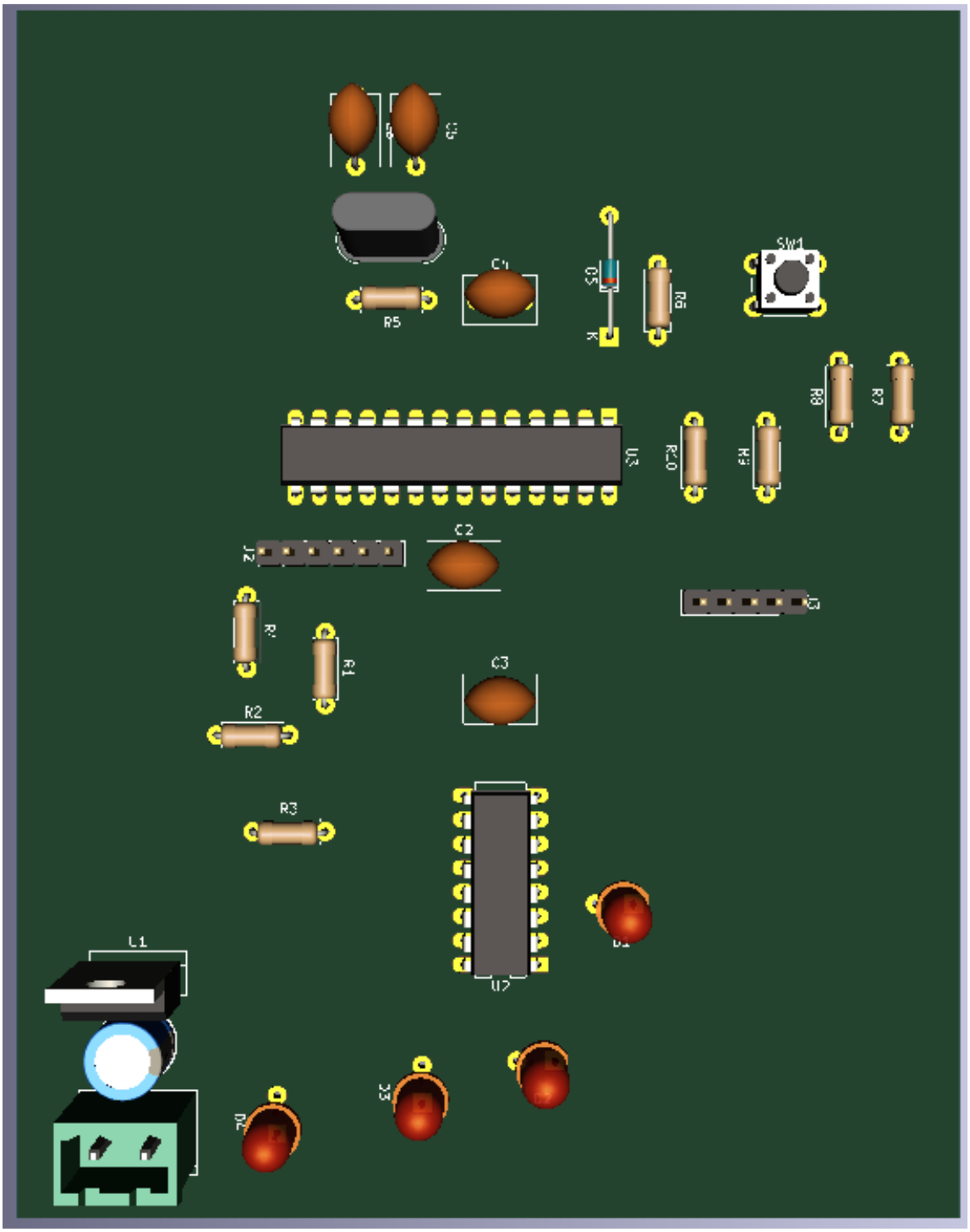

FIGURA XI - 14 – SIMULACIÓN DE LA CARA TOP (CARA DE COMPONENTES) DEL NODO DE GESTIÓN DE LUCES

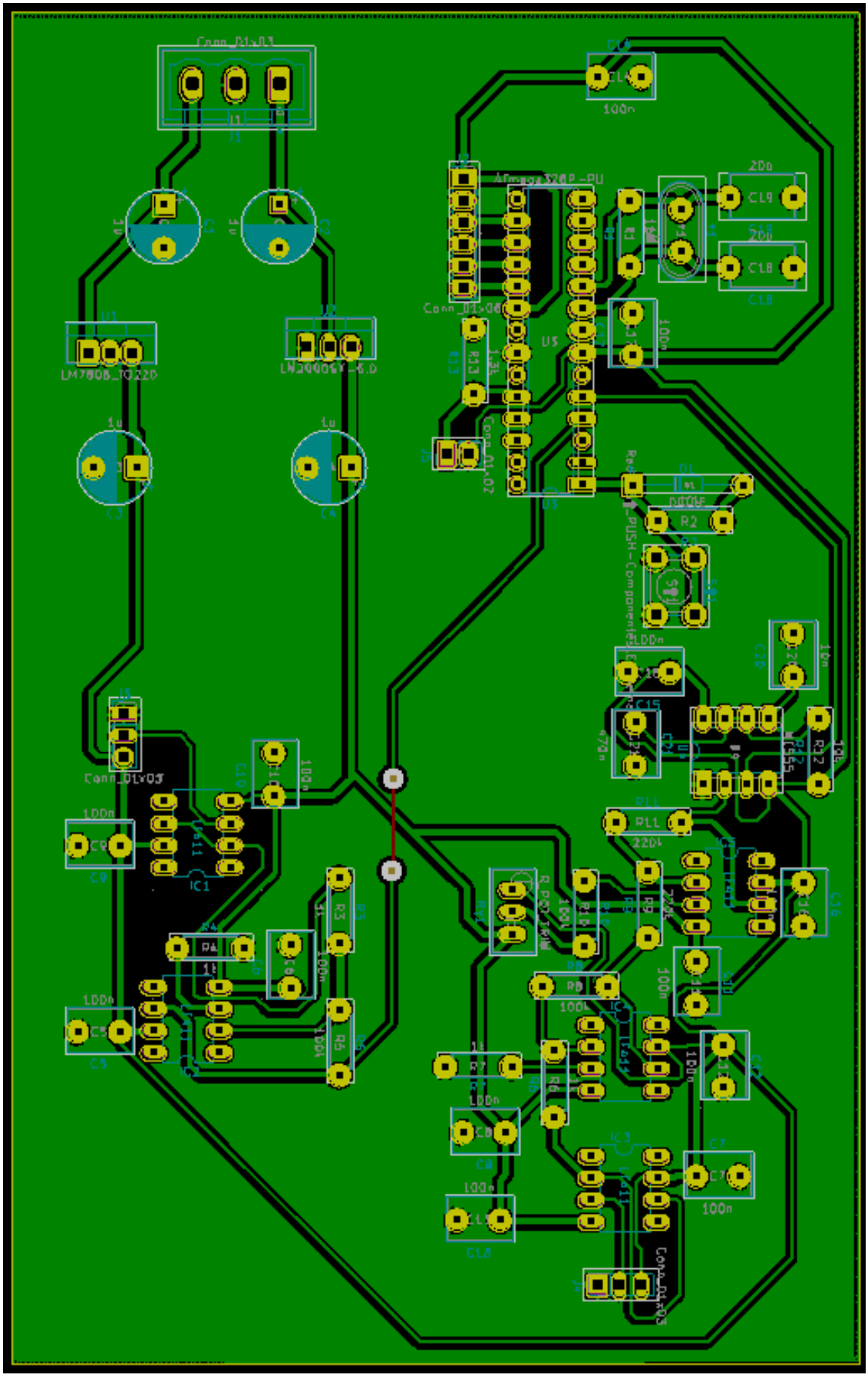

FIGURA XI - 15 – CARA BOTTOM DEL NODO EMISOR

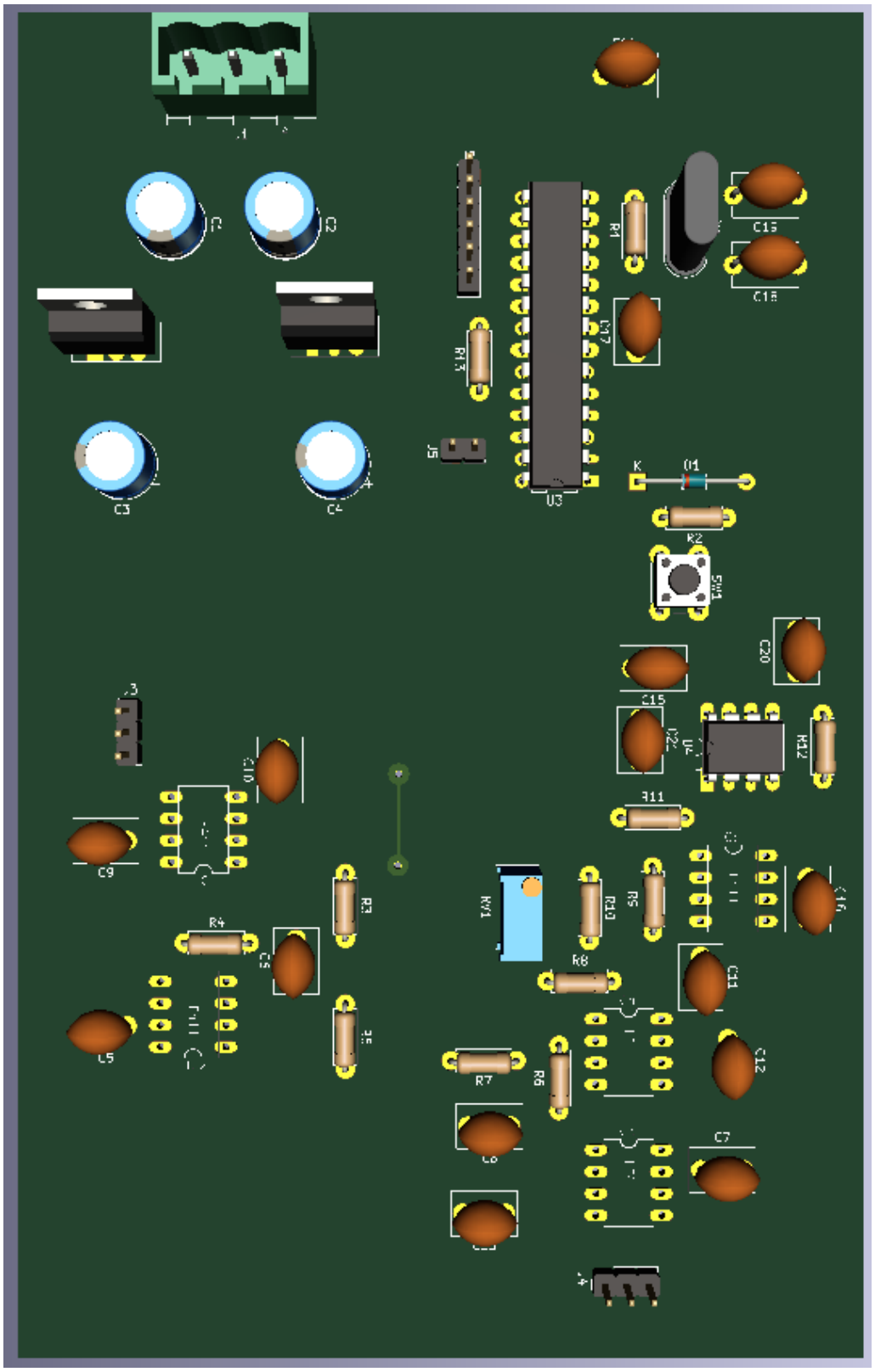

FIGURA XI - 16 – SIMULACIÓN DE LA CARA TOP (CARA DE COMPONENTES) DEL NODO EMISOR

XI.2.3. Fabricación

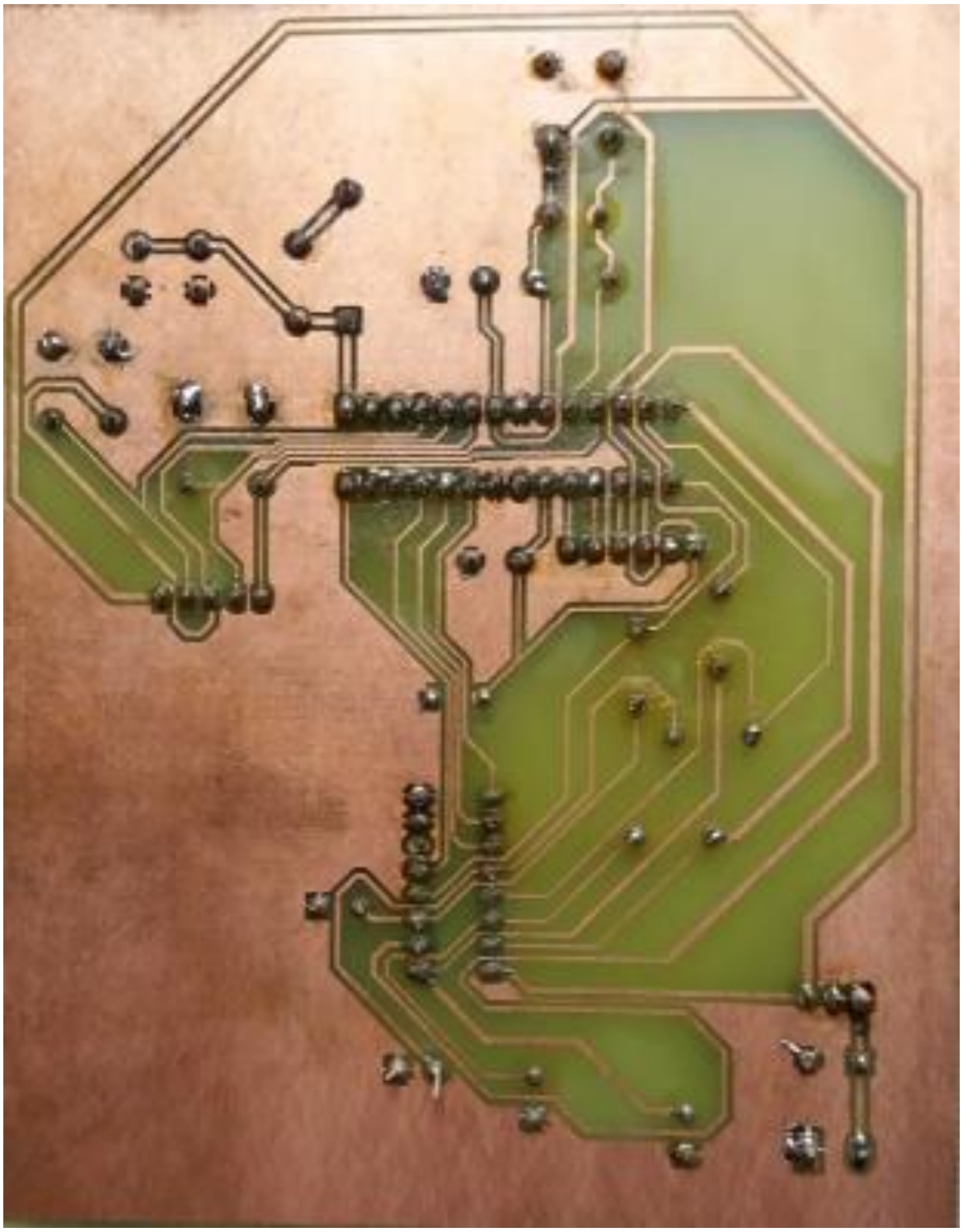

FIGURA XI - 17 - CARA BOTTOM REAL DEL NODO DE GESTIÓN DE LUCES

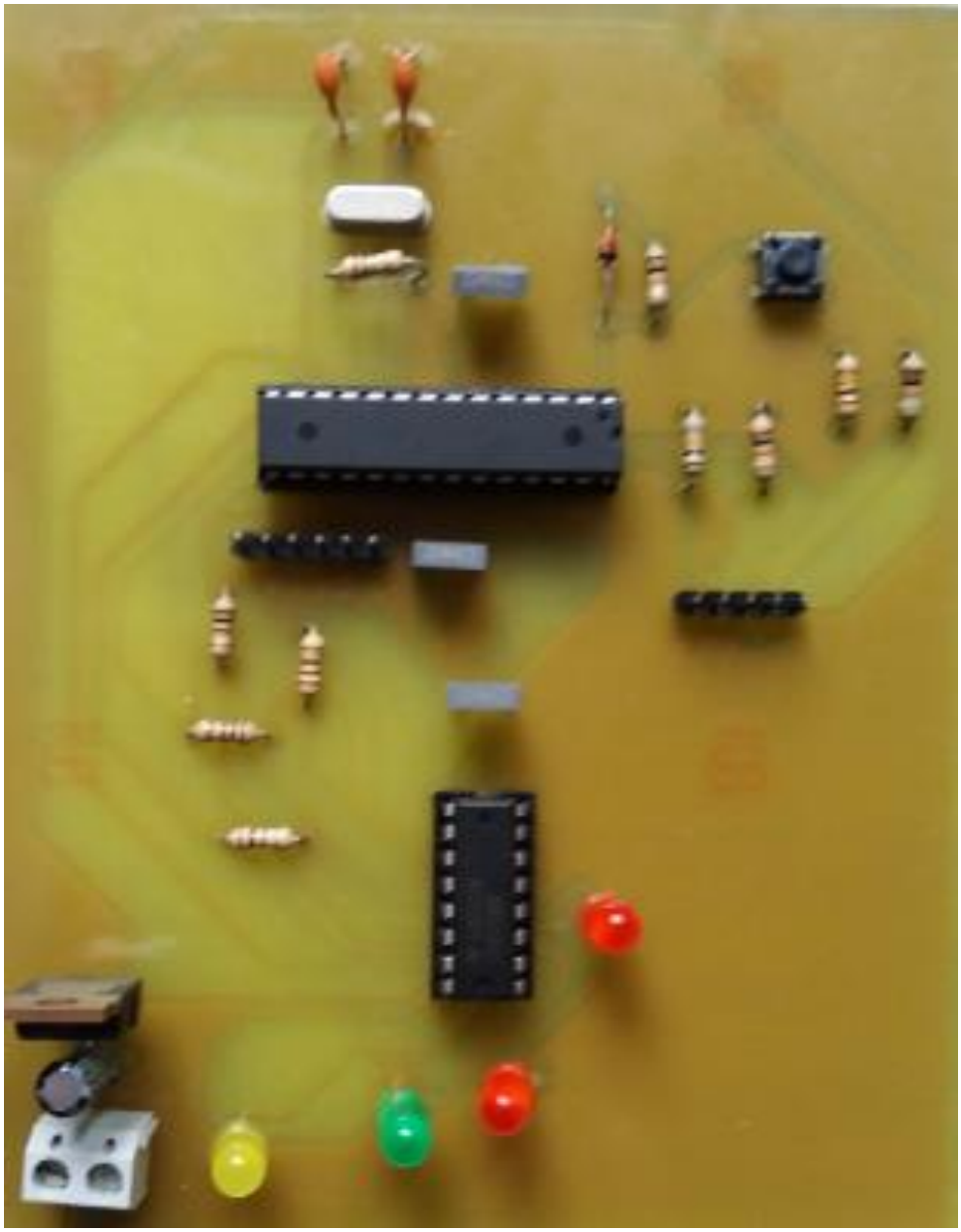

FIGURA XI - 18 - CARA TOP REAL (CARA DE COMPONENTES) DEL NODO DE GESTIÓN DE LUCES

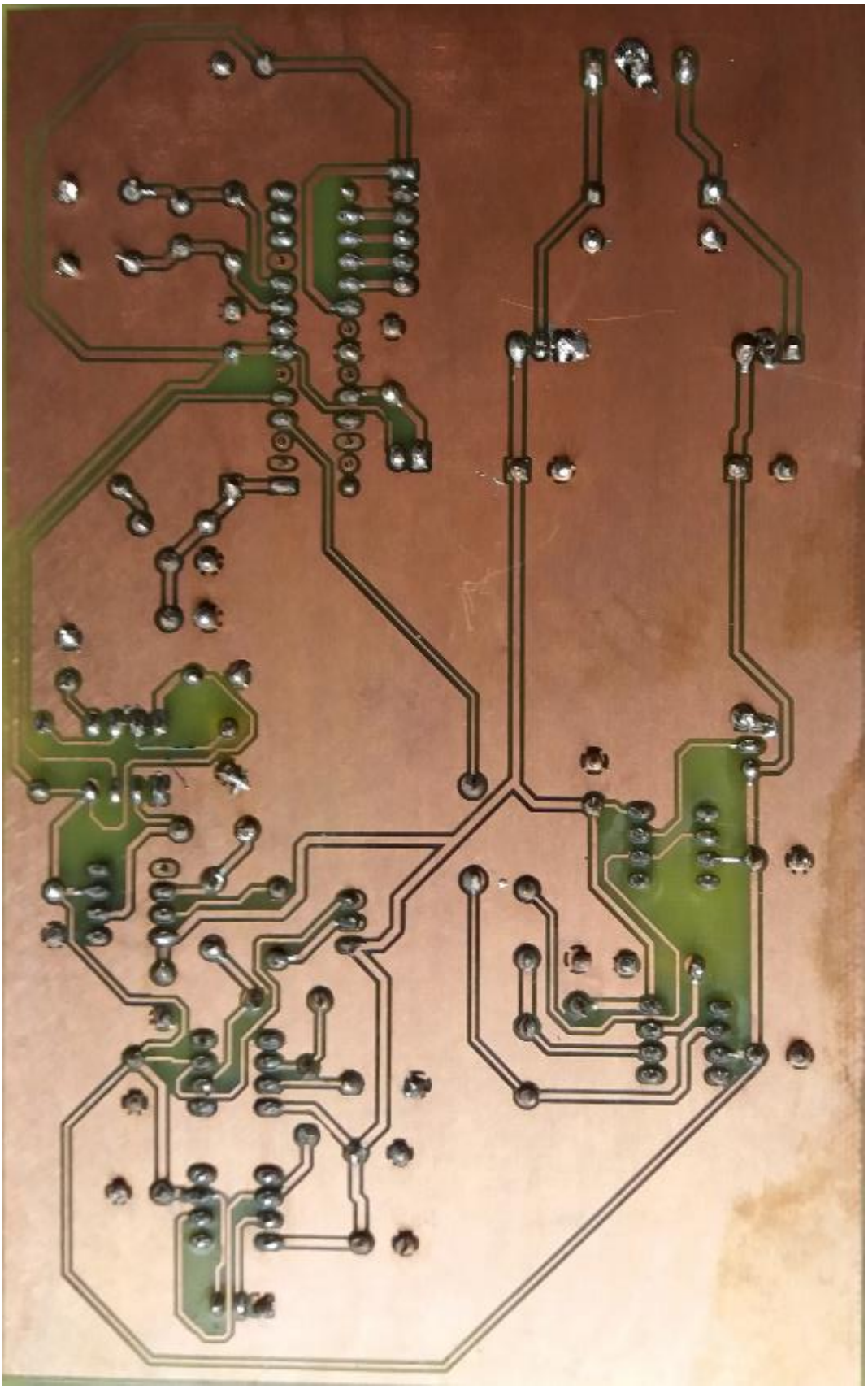

FIGURA XI - 19 – CARA BOTTOM REAL DEL NODO DE GESTIÓN DE LUCES

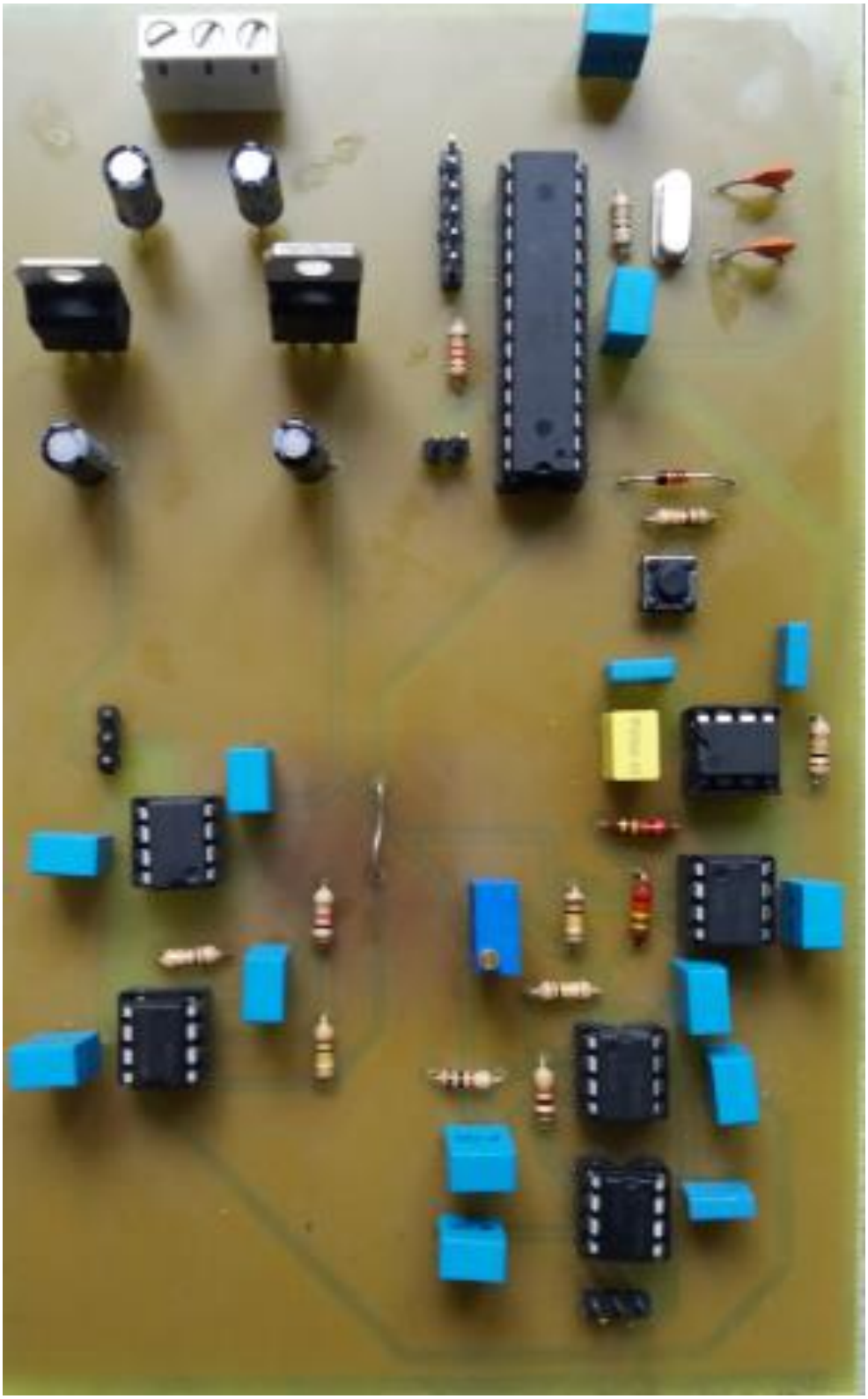

FIGURA XI - 20 – CARA TOP REAL (CARA DE COMPONENTES) DEL NODO EMISOR

XI.3. Código

XI.3.1. Nodo de emisión constante

```
#include <can.h>
#include <mcp2515.h>
#include <SPI.h>
#define PI 3.14159265358979323846264338327795 //Para calcular la longitud de la circunferencia de la rueda
#define diametroRueda (0.30)
#define numInterrupciones1 (7) //Define cuantas veces se va a ver interrumpido el haz en una sola vuelta del eje del motor
#define numInterrupciones2 (7) //Lo mismo pero en la rueda
#define Vcc (5) //Necesario para entender las lecturas analógicas


//Declaración de mensajes CAN
struct can_frame revoluciones1; //Revoluciones a la salida del motor
struct can_frame revoluciones2; //Revoluciones en la rueda
struct can_frame velocidad; //Velocidad real del coche; calculada a partir de las revoluciones en la rueda
struct can_frame temperatura;
MCP2515 mcp2515(10); //Pin del SS(Slave Select) del SPI


//Declaración de variables globales
int cuentaRPM1 = 0;
int cuentaRPM2 = 0;
int _rpm1 = 0;
int _rpm2 = 0;
int _velocidad = 0;
int _temp = 0;
```

```
void setup() {

  //Iniciaización del serial, SPI y CAN
  Serial.begin(9600);
  SPI.begin();
  mcp2515.reset();
  mcp2515.setBitrate(CAN_125KBPS, MCP_8MHZ);
  mcp2515.setNormalMode();

  //Declaración de IDs y DLCs de los datos
  velocidad.can_id = 0xA1;
  velocidad.can_dlc = 1; //Se puede codificar hasta 255 km/h
  revoluciones1.can_id = 0xA2;
  revoluciones1.can_dlc = 2; //Se pueden codificar hasta 65535 rpm
  revoluciones2.can_id = 0xA3;
  revoluciones2.can_dlc = 2; //Se pueden codificar hasta 65535 rpm
  temperatura.can_id = 0xA4;
  temperatura.can_dlc = 1; //Se pueden codificar hasta 255ºC


  pinMode(2, INPUT); //INT0
  pinMode(3, INPUT); //INT1
  pinMode(A1, INPUT); //Lectura de la NTC (temperatura)

}
```

```
void loop() {

  calculateRPM();
  calculateTemp();

  startMessage();

}


void calculateRPM(){

    detachInterrupt(digitalPinToInterrupt(2)); //Pin de interrupción INT0
    detachInterrupt(digitalPinToInterrupt(3)); //Pin de interrupción INT1
    attachInterrupt(digitalPinToInterrupt(2), readRPM1, RISING);
    attachInterrupt(digitalPinToInterrupt(3), readRPM2, RISING);
    delay(1000);
    detachInterrupt(digitalPinToInterrupt(2));
    detachInterrupt(digitalPinToInterrupt(3));

    /*
    Serial.print("Frecuencia1: ");
    Serial.println(cuentaRPM1);
    Serial.print("Frecuencia2: ");
    Serial.println(cuentaRPM2);
    */
    _rpm1 = cuentaRPM1/numInterrupciones1 * 60;//Pasa de rev/s(frecuencia) a rpm
    _rpm2 = cuentaRPM2/numInterrupciones2 * 60;
    _velocidad = _rpm2*PI*diametroRueda*60/1000; //Calculo de velocidad en km/h

    cuentaRPM1 = 0; //Reinicia la cuenta
    cuentaRPM2 = 0;
```

```
}

void readRPM1(){
  cuentaRPM1++;
}
void readRPM2(){
  cuentaRPM2++;
}


void calculateTemp(){

  float VxD = analogRead(A1);
  float Vx = VxD*Vcc/1024;
  float Rx = (1300*Vcc/Vx) - 1300;
  _temp = (-1/0.032)*log(Rx/4606.2);
  /*
  Serial.print("Tª NTC: ");
  Serial.println(T);
  */
}

void startMessage(){

  String revoluciones1Data0;
  String revoluciones1Data1;
  String revoluciones2Data0;
  String revoluciones2Data1;
  String velocidadData0;
  String temperaturaData0;
```

```
  int _rpm1Aux = _rpm1; //Para no modificar los datos originales mientras se hace el
cálculo
  int _rpm2Aux = _rpm2;
  int _velocidadAux = _velocidad;
  int _tempAux = _temp;

  toHex2byte(revoluciones1Data0, revoluciones1Data1, _rpm1Aux);
  toHex2byte(revoluciones2Data0, revoluciones2Data1, _rpm2Aux);
  toHex1byte(velocidadData0, _velocidadAux);
  toHex1byte(temperaturaData0, _tempAux);



  //Conversión de los String a Char
  int rev1_data0_len = revoluciones1Data0.length() + 1;
  char r1d0[rev1_data0_len] = {};
  revoluciones1Data0.toCharArray(r1d0, rev1_data0_len);
  int rev1_data1_len = revoluciones1Data1.length() + 1;
  char r1d1[rev1_data1_len] = {};
  revoluciones1Data1.toCharArray(r1d1, rev1_data1_len);

  int rev2_data0_len = revoluciones2Data0.length() + 1;
  char r2d0[rev2_data0_len] = {};
  revoluciones2Data0.toCharArray(r2d0, rev2_data0_len);
  int rev2_data1_len = revoluciones2Data1.length() + 1;
  char r2d1[rev2_data1_len] = {};
  revoluciones2Data1.toCharArray(r2d1, rev2_data1_len);

  int vel0_len = velocidadData0.length() + 1;
  char v0[vel0_len] = {};
  velocidadData0.toCharArray(v0,vel0_len);

  int temp0_len = temperaturaData0.length() + 1;
```

```
  char t0[temp0_len] = {};
  temperaturaData0.toCharArray(t0,temp0_len);

  //Conversión de los Char a enteros en Hexadecimal
  int number1 = (int)strtol(r1d0, NULL, 16);
  int number2 = (int)strtol(r1d1, NULL, 16);
  int number3 = (int)strtol(v0, NULL, 16);
  int number4 = (int)strtol(t0, NULL, 16);
  int number5 = (int)strtol(r2d0, NULL, 16);
  int number6 = (int)strtol(r2d1, NULL, 16);

  revoluciones1.data[0] = number1;
  revoluciones1.data[1] = number2;
  velocidad.data[0] = number3;
  temperatura.data[0] = number4;
  revoluciones2.data[0] = number5;
  revoluciones2.data[1] = number6;

  sendMessage(1);
  sendMessage(2);
  sendMessage(3);
  sendMessage(4);

}

void sendMessage(int a){ //Para poder enviar los mensajes con el delay(100), sin que
ocurra error de segmentación
  switch(a){
    case 1: mcp2515.sendMessage(&revoluciones1); break;
    case 2: mcp2515.sendMessage(&revoluciones2); break;
    case 3: mcp2515.sendMessage(&velocidad); break;
    case 4: mcp2515.sendMessage(&temperatura); break;
```

```
  }
  delay(100);
}

void toHex2byte(String &Data0, String &Data1, int _dataValor){

  //CONVERSIÓN DE 2 BYTE
  int interacciones = 0;
  int resto = 0;
  char aux1[10]={}; //Da problemas cuando se le asigna 4 justo

  if(_dataValor >= 4096){
    interacciones = 4;
  }
  else if(_dataValor >= 256 && _dataValor <= 4095){
    interacciones = 3;
  }
  else if(_dataValor >= 16 && _dataValor <= 255){
    interacciones = 2;
  }

     for(int i = 0; i < interacciones; i++){
     resto = _dataValor % 16;
       switch(resto){
         case 0: aux1[i] = '0'; break;
         case 1: aux1[i] = '1'; break;
         case 2: aux1[i] = '2'; break;
         case 3: aux1[i] = '3'; break;
         case 4: aux1[i] = '4'; break;
         case 5: aux1[i] = '5'; break;
         case 6: aux1[i] = '6'; break;
         case 7: aux1[i] = '7'; break;
```

```
        case 8: aux1[i] = '8'; break;
        case 9: aux1[i] = '9'; break;
        case 10: aux1[i] = 'A'; break;
        case 11: aux1[i] = 'B'; break;
        case 12: aux1[i] = 'C'; break;
        case 13: aux1[i] = 'D'; break;
        case 14: aux1[i] = 'E'; break;
        case 15: aux1[i] = 'F'; break;
       }
    _dataValor = _dataValor / 16;
   }


 //aux1[interacciones] = _rpm / 16; //En el último dígito se almacena el cociente, no el
resto
  if(interacciones == 3){
    aux1[3] = '0';
  }
  else if(interacciones == 2){
    aux1[2] = '0';
    aux1[3] = '0';
  }


  //HAY QUE INVERTIR EL CHAR, SE HA ALMACENADO A LA INVERSA
  char aux2[10]={};
  for (int i = 0, k = 3; i < 4; i++, k--) {
    aux2[k] = aux1[i];
  }


  //SE SEPARA EL CHAR EN 2, PARA ADAPTARSE AL FORMATO BUS-CAN
```

```
  char auxData0[10]={};
  for(int i = 0; i < 2; i++){ //Si aux2 = '1000'
    auxData0[i] = aux2[i]; //revData0 seria '10'
  }


  char auxData1[10]={};
  for(int i = 2; i < 4; i++){ //revData1 seria '00'
    auxData1[i - 2] = aux2[i];
  }
  String strData0 = String(auxData0);
  Data0 = strData0; //SE ALMACENA EN LA VARIABLE QUE SALE DE LA FUNCIÓN
  Data0 = "0x" + Data0; //SE LE AÑADE FORMATO
  String strData1 = String(auxData1);
  Data1 = strData1; //SE ALMACENA EN LA VARIABLE QUE SALE DE LA FUNCIÓN
  Data1 = "0x" + Data1; //SE LE AÑADE FORMATO

}

void toHex1byte(String &Data0, int _dataValor){

  if(_dataValor <= 15){
    char aux1[10]={};
    switch(_dataValor){
        case 0: aux1[0] = '0'; break;
        case 1: aux1[0] = '1'; break;
        case 2: aux1[0] = '2'; break;
        case 3: aux1[0] = '3'; break;
        case 4: aux1[0] = '4'; break;
        case 5: aux1[0] = '5'; break;
        case 6: aux1[0] = '6'; break;
        case 7: aux1[0] = '7'; break;
```

```
      case 8: aux1[0] = '8'; break;
      case 9: aux1[0] = '9'; break;
      case 10: aux1[0] = 'A'; break;
      case 11: aux1[0] = 'B'; break;
      case 12: aux1[0] = 'C'; break;
      case 13: aux1[0] = 'D'; break;
      case 14: aux1[0] = 'E'; break;
      case 15: aux1[0] = 'F'; break;
    }
    String strData0 = String(aux1);
    Data0 = strData0;
    Data0 = "0x" + Data0;
}

else{
  int interacciones = 2; //Sólo es un byte, son 2 números
  int resto = 0;
  char aux1[10]={};

  for(int i = 0; i < interacciones; i++){
  resto = _dataValor % 16;
    switch(resto){
      case 0: aux1[i] = '0'; break;
      case 1: aux1[i] = '1'; break;
      case 2: aux1[i] = '2'; break;
      case 3: aux1[i] = '3'; break;
      case 4: aux1[i] = '4'; break;
      case 5: aux1[i] = '5'; break;
      case 6: aux1[i] = '6'; break;
      case 7: aux1[i] = '7'; break;
      case 8: aux1[i] = '8'; break;
      case 9: aux1[i] = '9'; break;
```

```
      case 10: aux1[i] = 'A'; break;
      case 11: aux1[i] = 'B'; break;
      case 12: aux1[i] = 'C'; break;
      case 13: aux1[i] = 'D'; break;
      case 14: aux1[i] = 'E'; break;
      case 15: aux1[i] = 'F'; break;
     }
    _dataValor = _dataValor / 16;

    }

    //HAY QUE INVERTIR EL CHAR, SE HA ALMACENADO A LA INVERSA
    char aux2[10] = {};
    for (int i = 0, k = 1; i < 2; i++, k--) {
     aux2[k] = aux1[i];
    }
    String strData0 = String(aux2);
    Data0 = strData0; //SE ALMACENA EN LA VARIABLE QUE SALE DE LA FUNCIÓN
    Data0 = "0x" + Data0; //SE LE AÑADE FORMATO
   }

}
```

XI.3.2. Nodo de gestión de luces

```
#include <can.h>
#include <mcp2515.h>
#include <SPI.h>


struct can_frame ledError;
MCP2515 mcp2515(10);

#define inPin1 (2) //Luces de cruce + posicion
#define inPin2 (3) //Solo posición
#define inPin3 (4) //Luces largas + posición
#define inPin4 (5) //Luces antiniebla + posición
#define ledPinRed (6) //Luz cruce
#define ledPinBlue (7) //Luz larga
#define ledPinYellow (8) //Luz antiniebla
#define ledPinGreen (9) //Luz posición


bool cruce = false;
bool posicion = false;
bool antiniebla = false;
bool larga = false;
int errorVector[4] = {0, 0, 0, 0};
int pastError = 0;
int currentError = 0;


void setup() {
  Serial.begin(9600);
  pinMode(ledPinRed, OUTPUT);
```

```
  pinMode(ledPinBlue, OUTPUT);
  pinMode(ledPinYellow, OUTPUT);
  pinMode(ledPinGreen, OUTPUT);
  pinMode(inPin1, INPUT);
  pinMode(inPin2, INPUT);
  pinMode(inPin3, INPUT);
  pinMode(inPin4, INPUT);

  pinMode(A0, OUTPUT); //Para seleccionar la salida del multiplexor
  pinMode(A1, OUTPUT);
  pinMode(A2, OUTPUT);
  pinMode(A5, INPUT); //Para recibir la salida del multiplexor

  //Inicialización BUS-CAN
  SPI.begin();
  mcp2515.reset();
  mcp2515.setBitrate(CAN_125KBPS, MCP_8MHZ);
  mcp2515.setNormalMode();
  ledError.can_id = 0x05; //Error importante
  ledError.can_dlc = 4;
  ledError.data[0] = 0x00; //Error cruce
  ledError.data[1] = 0x00; //Error posición
  ledError.data[2] = 0x00; //Error antiniebla
  ledError.data[3] = 0x00; //Error larga

}

void loop() {

  //Luces de cruce + posicion
  if(digitalRead(inPin1) == 1){
    digitalWrite(ledPinRed, HIGH);
```

```
  digitalWrite(ledPinGreen, HIGH);
  digitalWrite(ledPinYellow, LOW);
  digitalWrite(ledPinBlue, LOW);
  cruce = true;
  posicion = true;
  antiniebla = false;
  larga = false;
 }
 //Solo posición
 else if(digitalRead(inPin2) == 1){
  digitalWrite(ledPinRed, LOW);
  digitalWrite(ledPinGreen, HIGH);
  digitalWrite(ledPinYellow, LOW);
  digitalWrite(ledPinBlue, LOW);
  cruce = false;
  posicion = true;
  antiniebla = false;
  larga = false;
 }
 //Luces largas + posición
 else if(digitalRead(inPin3) == 1){
  digitalWrite(ledPinRed, LOW);
  digitalWrite(ledPinGreen, HIGH);
  digitalWrite(ledPinYellow, LOW);
  digitalWrite(ledPinBlue, HIGH);
  cruce = false;
  posicion = true;
  antiniebla = false;
  larga = true;
 }
 //Luces antiniebla + posición
 else if(digitalRead(inPin4) == 1){
```

```
  digitalWrite(ledPinRed, LOW);
  digitalWrite(ledPinGreen, HIGH);
  digitalWrite(ledPinYellow, HIGH);
  digitalWrite(ledPinBlue, LOW);
  cruce = false;
  posicion = true;
  antiniebla = true;
  larga = false;
}
else{
  digitalWrite(ledPinRed, LOW);
  digitalWrite(ledPinGreen, LOW);
  digitalWrite(ledPinYellow, LOW);
  digitalWrite(ledPinBlue, LOW);
  cruce = false;
  posicion = false;
  antiniebla = false;
  larga = false;
}

checkLight(); //Revisa el estado de las luces, si hay alguna fundida dice cual es
for(int i = 0; i < 4; i++){
  switch(i){
    case 0:
      Serial.print("CRUCE: ");
      if(errorVector[i] == 0){
        Serial.println("OK");
        ledError.data[0] = 0x00;
      }
      else{
        Serial.println("ERROR");
        ledError.data[0] = 0x01;
```

```
    }; break;
  case 1:
    Serial.print("POSICIÓN: ");
    if(errorVector[i] == 0){
      Serial.println("OK");
      ledError.data[1] = 0x00;
    }
    else{
      Serial.println("ERROR");
      ledError.data[1] = 0x01;
    }; break;
  case 2:
    Serial.print("ANTINIEBLA: ");
    if(errorVector[i] == 0){
      Serial.println("OK");
      ledError.data[2] = 0x00;
    }
    else{
      Serial.println("ERROR");
      ledError.data[2] = 0x01;
    }; break;
  case 3:
    Serial.print("LARGA: ");
    if(errorVector[i] == 0){
      Serial.println("OK");
      ledError.data[3] = 0x00;
    }
    else{
      Serial.println("ERROR");
      ledError.data[3] = 0x01;
    }; break;
}
```

```
  }


  for(int i = 0; i < 4; i++){
    if(ledError.data[i] == 0x01){
      currentError++; //Desde que haya un solo error, "currentError" será mayor que
cero
    }
  }

  if(currentError != pastError){
    sendMessage(); //Sólo hace falta mandar un mensaje si el dato ha cambiado, así no
se satura el canal
  }
  pastError = currentError; //Se actualiza el valor del error para la siguiente interacción
  currentError = 0; //Se reinicia el error para que no se acumule en la siguiente
interacción*/

}

void sendMessage(){
  for(int i = 0; i < 10; i++){
    mcp2515.sendMessage(&ledError);
    delay(200);
    Serial.println("Mensaje enviado");
  }
}

void checkLight(){

  //Comprobación luz de cruce
  digitalWrite(A0, LOW); // 0
```

```
  digitalWrite(A1, LOW);  // 0
  digitalWrite(A2, LOW);  // 0
  int highCruce = analogRead(A5);
  digitalWrite(A0, LOW);   //  0
  digitalWrite(A1, LOW);  //   0
  digitalWrite(A2, HIGH);   // 1
  int lowCruce = analogRead(A5);
  if(highCruce > (lowCruce + 20) && cruce == true){ //MARGEN DE SEGURIDAD
NECESARIO
    Serial.println("CRUCE: OK");
    errorVector[0] = 0;
  }
  else if(cruce == true){
    Serial.println("CRUCE: ERROR");
    errorVector[0] = 1;
  }


  //Comprobación luz de posición
  digitalWrite(A0, LOW); //   0
  digitalWrite(A1, HIGH);  // 1
  digitalWrite(A2, LOW);  //  0
  int highPosicion = analogRead(A5);
  digitalWrite(A0, LOW);   //   0
  digitalWrite(A1, HIGH);  //   1
  digitalWrite(A2, HIGH);   //  1
  int lowPosicion = analogRead(A5);
  if(highPosicion > (lowPosicion + 20) && posicion == true){
    Serial.println("POSICIÓN: OK");
    errorVector[1] = 0;
  }
  else if(posicion == true){
```

```
  Serial.println("POSICIÓN: ERROR");
  errorVector[1] = 1;
}

//Comprobación luz antiniebla
digitalWrite(A0, HIGH); // 1
digitalWrite(A1, LOW);  // 0
digitalWrite(A2, LOW); //  0
int highAntiN = analogRead(A5);
digitalWrite(A0, HIGH); // 1
digitalWrite(A1, LOW); //  0
digitalWrite(A2, HIGH); // 1
int lowAntiN = analogRead(A5);
if(highAntiN > (lowAntiN + 20) && antiniebla == true){
  Serial.println("ANTINIEBLA: OK");
  errorVector[2] = 0;
}
else if(antiniebla == true){
  Serial.println("ANTINIEBLA: ERROR");
  errorVector[2] = 1;
}

//Comprobación luz larga
digitalWrite(A0, HIGH); //1
digitalWrite(A1, HIGH); //1
digitalWrite(A2, LOW); // 0
int highLarga = analogRead(A5);
digitalWrite(A0, HIGH); // 1
digitalWrite(A1, HIGH); // 1
digitalWrite(A2, HIGH); // 1
int lowLarga = analogRead(A5);
if(highLarga > (lowLarga + 20) && larga == true){
```

```
    Serial.println("LARGA: OK");
    errorVector[3] = 0;
  }
  else if(larga == true){
    Serial.println("LARGA: ERROR");
    errorVector[3] = 1;
  }

}
```

XI.3.3. Nodo *Sniffer*

```
#include <SPI.h>
#include <can.h>
#include <mcp2515.h>

struct can_frame mensajeGeneral;
struct can_frame revoluciones1;
struct can_frame revoluciones2;
struct can_frame velocidad;
struct can_frame temperatura;
struct can_frame ledError;
MCP2515 mcp2515(10);



void setup() {
  Serial.begin(9600);
  SPI.begin();

  mcp2515.reset();
  mcp2515.setBitrate(CAN_125KBPS, MCP_8MHZ);
  mcp2515.setNormalMode();
  Serial.println("----------------CAN READ------------");

  //Inicialización a 0 de los datos de las variables CAN
  velocidad.can_dlc = 1;
  velocidad.data[0] = 0x00;
  revoluciones1.can_dlc = 2;
  revoluciones1.data[0] = 0x00;
  revoluciones1.data[1] = 0x00;
  revoluciones2.can_dlc = 2;
  revoluciones2.data[0] = 0x00;
```

```
  revoluciones2.data[1] = 0x00;
  temperatura.can_dlc = 1;
  temperatura.data[0] = 0x00;
  ledError.can_dlc = 4;
  ledError.data[0] = 0x01;
  ledError.data[1] = 0x01;
  ledError.data[2] = 0x01;
  ledError.data[3] = 0x01;

}

void loop() {

  if(mcp2515.readMessage(&mensajeGeneral)){
     String comparacion = String(mensajeGeneral.can_id);

     if(comparacion == "161"){//VELOCIDAD
       velocidad.can_id = mensajeGeneral.can_id;
       velocidad.data[0] = mensajeGeneral.data[0];
     }
     else if(comparacion == "162"){//REVOLUCIONES1
       revoluciones1.can_id = mensajeGeneral.can_id;
       revoluciones1.data[0] = mensajeGeneral.data[0];
       revoluciones1.data[1] = mensajeGeneral.data[1];
     }
     else if(comparacion == "163"){//REVOLUCIONES2
       revoluciones2.can_id = mensajeGeneral.can_id;
       revoluciones2.data[0] = mensajeGeneral.data[0];
       revoluciones2.data[1] = mensajeGeneral.data[1];
     }
     else if(comparacion == "164"){//TEMPERATURA
       temperatura.can_id = mensajeGeneral.can_id;
```

```
  temperatura.data[0] = mensajeGeneral.data[0];
  temperatura.data[1] = mensajeGeneral.data[1];
}
else if(comparacion == "5"){//LUCES
  ledError.can_id = mensajeGeneral.can_id;
  ledError.data[0] = mensajeGeneral.data[0];
  ledError.data[1] = mensajeGeneral.data[1];
  ledError.data[2] = mensajeGeneral.data[2];
  ledError.data[3] = mensajeGeneral.data[3];
}
Serial.print("Velocidad: ");
    for(int i = 0; i < velocidad.can_dlc; i++){
      Serial.println(velocidad.data[i],HEX);
     }

Serial.print("Revoluciones1: ");
      for(int i = 0; i < revoluciones1.can_dlc; i++){
        Serial.print(revoluciones1.data[i],HEX);
        Serial.print(" ");
      }
Serial.print("Revoluciones2: ");
      for(int i = 0; i < revoluciones2.can_dlc; i++){
        Serial.print(revoluciones2.data[i],HEX);
        Serial.print(" ");
      }
      Serial.println();
Serial.print("Temperatura: ");
      for(int i = 0; i < temperatura.can_dlc; i++){
        Serial.print(temperatura.data[i],HEX);
        Serial.println(" ");
      }
```

```
for(int i = 0; i < 4; i++){
  switch(i){
    case 0:
      Serial.print("CRUCE: ");
      if(ledError.data[0] == 0x00){
        Serial.println("OK");
      }
      else{
        Serial.println("ERROR");
      }; break;
    case 1:
      Serial.print("POSICIÓN: ");
      if(ledError.data[1] == 0x00){
        Serial.println("OK");
      }
      else{
        Serial.println("ERROR");
      }; break;
    case 2:
      Serial.print("ANTINIEBLA: ");
      if(ledError.data[2] == 0x00){
        Serial.println("OK");
      }
      else{
        Serial.println("ERROR");
      }; break;
    case 3:
      Serial.print("LARGA: ");
      if(ledError.data[3] == 0x00){
        Serial.println("OK");
      }
      else{
```

```
            Serial.println("ERROR");
          }; break;
        }
      }
      Serial.println();
  }



}
```