Trabajo fin de Grado

Universidad de Valladolid

Grado en Ingeniería de Tecnologías de Telecomunicación

Interfaz WiFi - CAN

WiFi - CAN Interface

Autor:
David Manso Fernández

Tutor:
Jesús Manuel Hernández Mangas

Mayo 2023

Título:	**Interfaz WiFi - CAN**
Autor:	**David Manso Fernández**
Tutor:	**Jesús Manuel Hernández Mangas**
Departamento:	**Electrónica**

Tribunal

Presidente:	**Jesús Arias Álvarez**
Secretario:	**Martín Jaraíz Maldonado**
Vocal:	**Jesús Manuel Hernández Mangas**

Fecha:	
Calificación:	

Agradecimientos

Quiero dedicar este proyecto a las personas que han sido fundamentales en mi vida y en el desarrollo de este proyecto:

Primeramente dar las gracias a mi profesor Jesús M. Hernández Mangas, su valiosa ayuda así como sus consejos y puntos de vista me han servido de gran utilidad.

A mis padres, Ramiro y Lola, les agradezco de todo corazón por inculcarme valores sólidos que han sido la base de mi formación. Su amor, apoyo y dedicación han sido fundamentales en cada paso que he dado.

A mis hermanos, Jaime y María, les agradezco su apoyo incondicional. Sus consejos, lecciones y el constante aliento que me han brindado han sido de gran importancia para alcanzar el punto en el que me encuentro ahora.

A mi pareja, Irene, no puedo expresar suficientemente mi gratitud. Su ayuda, comprensión y apoyo inquebrantable han sido pilares indispensables en los momentos más difíciles a lo largo de este proyecto. Sin ella, no hubiera sido posible superar los desafíos que se presentaron en el camino.

A mis compañeros de clase, les agradezco sinceramente su colaboración. Sus aportes, ideas y perspectivas diferentes han enriquecido enormemente este proyecto, brindándome una visión más amplia y diversa.

A mis amigos, quiero expresar mi gratitud por su constante ánimo y confianza en mí. Su apoyo incondicional ha sido un motor fundamental en cada etapa de este proyecto, dándome fuerzas para perseverar y superar los obstáculos que surgieron en el camino.

A todas estas personas, mi más profundo agradecimiento. Su influencia, guía y apoyo han sido esenciales en mi crecimiento personal y profesional. Sin vosotros, este proyecto no habría sido posible.

Resumen

Este proyecto presenta el diseño de un protocolo de comunicación basado en CAN con el objetivo de superar las limitaciones existentes en la transmisión de información de gran tamaño, así como proporcionar conectividad a Internet a los dispositivos que carecen de ella y que, además, solo disponen de conexión a un bus CAN. El protocolo desarrollado se encarga de gestionar las tramas enviadas tanto por el cliente como por el servidor, al igual que de asegurar la cohesión de múltiples tramas con el fin de transmitir información de forma unificada. A mayores, se ha construido un prototipo que incluye los componentes necesarios para lograr los objetivos establecidos, siendo el módulo ESP12-F con el microprocesador ESP8266 su núcleo principal.

Palabras Clave: Protocolo CAN, ESP12-F, ESP8266, LPC2103, Protocolo de Comunicación, MCP2515.

Abstract

This project presents the design of a communication protocol based on CAN with the aim of overcoming the existing limitations in the transmission of large information, as well as providing Internet connectivity to devices that lack it and that, in addition, only have a connection to a CAN bus. The developed protocol is responsible for managing the frames sent by both the client and the server, as well as ensuring the cohesion of multiple frames in order to transmit information in a unified way. In addition, a prototype has been built that includes the necessary components to achieve the established objectives, being the ESP12-F module with the ESP8266 microprocessor its main core.

Keywords: CAN Protocol, ESP12-F, ESP8266, LPC2103, Communication Protocol, MCP2515.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En la actualidad, numerosos dispositivos utilizan el protocolo CAN para la comunicación, especialmente en el ámbito de la automoción, aunque también tiene una gran relevancia en la industria en general. Estos componentes suelen formar parte de un sistema con topología de bus, donde todos los nodos están interconectados. Sin embargo, a pesar de ser económicos, la mayoría de estos nodos carecen de una capacidad computacional significativa.

Por tanto, se requiere un sistema capaz de realizar los cálculos y tareas necesarios en nombre de estos dispositivos, de manera que solo reciban la información mínima indispensable. Esto permitiría lograr una comunicación eficiente y evitar un exceso en la demanda de los recursos de estos componentes de bajo coste.

El protocolo CAN es ampliamente utilizado en las comunicaciones industriales y en la automoción, para la interconexión de las unidades de control de los vehículos. Por tanto, es deseable mantener este método de comunicación, pero dotar al sistema de la capacidad de recibir comunicaciones del exterior supondría un gran avance en los sistemas existentes. Gracias a este proyecto, se podrían proporcionar sistemas completos que se comunican a través de una topología en bus con acceso a redes externas, ya sea mediante conexión WiFi u otro tipo de conexión inalámbrica.

El objetivo de este trabajo de fin de grado es dotar de conexiones inalámbricas a dispositivos que actualmente carecen de ellas y que, además, solo disponen del protocolo CAN como medio de comunicación. Para ello, se han diseñado dos elementos que, aunque completamente diferentes, se integrarán para cumplir con los objetivos propuestos.

En primer lugar, se ha desarrollado un protocolo de comunicación que abarca desde el nivel más bajo hasta el más alto y que es capaz de solicitar, recibir y confirmar información utilizando el protocolo CAN. Cabe mencionar que este protocolo CAN solo permite la transmisión de hasta 8 bytes de información por trama. El protocolo diseñado abordará la estructura de los campos de una trama CAN, así como la distribución de tareas entre el cliente que solicita la información y el servidor que atiende las peticiones.

Además, se diseñará una aplicación, que sirva tanto para el cliente como para el servidor, que permita al primero solicitar diferentes tipos de archivos y al segundo obtenerlos desde un sistema de ficheros o descargarlos desde una página de internet. Para hacer posible esta funcionalidad, se ha diseñado un prototipo cuyo componente central es un ESP8266 que opera en el entorno de Arduino y cuenta con conectividad WiFi, así como la capacidad de integrar un sistema de ficheros. El prototipo contiene los componentes necesarios para poder comunicarse con el resto de dispositivos dentro del bus que no dispongan de conexión a internet mediante el protocolo CAN, permitiéndoles recibir información del exterior, actualizaciones vía OTA u otros usos que requieran acceso a internet o una carga computacional elevada.

Capítulo 2

Protocolo CAN

En la década de los años 80, ingenieros de la empresa **Robert Bosch GmbH** se dedicaron al estudio de la implementación de sistemas de bus seriales en automóviles para cumplir con los requisitos establecidos por la Sociedad de Ingenieros Automotrices (SAE, *Society of Automotive Engineers*). Como resultado de estos estudios, se desarrolló el **protocolo de comunicaciones CAN**, el cual proporcionaba una solución para la interconexión de sensores, actuadores y controladores en vehículos. [Morales and Antonio, 2005]

Los estándares RS-232 y RS-485 han sido ampliamente utilizados como interfaces para la transferencia de información entre ordenadores y controladores. Sin embargo, ninguno de estos estándares ni otros similares eran adecuados para su aplicación en vehículos debido a la complejidad de la arquitectura de red, son poco fiables y los elevados costes de implementación. La situación cambió con la estandarización del protocolo CAN.[Sánchez Vela et al., 2016]

La razón principal detrás de esta estandarización fue la significativa reducción en la complejidad del cableado de los sistemas de adquisición y procesamiento presentes en los vehículos modernos. Esto permitió lograr una interconexión más eficiente y fiable de los componentes del vehículo a través de una red de área local, satisfaciendo así las necesidades de comunicación en la industria automotriz.

2.1. Marco Histórico del Protocolo CAN

En 1983, la empresa Robert Bosch GmbH comenzó un proyecto interno con el objetivo de desarrollar una red de interconexión para los dispositivos electrónicos presentes en los automóviles, con el fin de mejorar su funcionalidad y reducir la cantidad de cableado necesario. Este proyecto fue liderado por **Siegfried Dais**, **Martin Litschel** y el **Dr. Uwe Kiencke**, con la contribución del **Dr. Wolfhard Lawrenz** como consultor. Fue el Dr. Lawrenz quien propuso el nombre de CAN (Controller Area Network) para este protocolo. [Morales and Antonio, 2005]

Durante el desarrollo de la tecnología CAN, se incorporaron al proyecto dos empresas líderes en sus respectivos sectores: **Mercedes-Benz**, fabricante de automóviles e **Intel Corp**, fabricante de microchips.

En la ciudad americana de Detroit se celebró el congreso de la SAE en febrero de 1986 donde Robert Bosch GmbH presentó oficialmente el sistema de bus serial CAN. Un año después, Intel lanzó la primera implementación física del protocolo con el **Controlador CAN 82526** mientras que la versión de Phillips llegó más tarde con el **Controlador CAN 82C200**, convirtiéndose en los pilares en los que se sentaron las bases para el desarrollo de los controladores actuales.

A pesar de que el bus CAN fue originalmente desarrollado para aplicaciones en automóviles, sus primeras implementaciones se llevaron a cabo en sectores de mercado no relacionados con la industria automotriz, especialmente en Europa. Algunos ejemplos de estas primeras aplicaciones son:

- La empresa finlandesa Kone utilizó este protocolo para la fabricación de ascensores.
- La división de sistemas médicos de la empresa holandesa Phillips, implementó una red interna para la comunicación de las máquinas de rayos X.

Durante la década de los noventa, se produjeron varios acontecimientos significativos para este protocolo que impulsaron su desarrollo, difusión y aplicación. A continuación, se detallan algunos de estos eventos:

- En 1991, Robert Bosch GmbH publicó la especificación CAN en su **versión 2.0 (CAN Specification 2.0)**. Posteriormente, esta especificación fue sometida a un proceso de estandarización internacional que se vio afectado por disputas políticas con el protocolo francés **VAN (Vehicle Area Network)**. [Morales and Antonio, 2005]
- En marzo de 1992, se creó la **CiA (CAN in Automation)**, una organización sin fines de lucro formada por fabricantes y usuarios internacionales. La misión de CiA era proporcionar información técnica de productos y promocionar el uso de CAN. Poco después de su fundación, CiA publicó un artículo técnico que recomendaba el uso de transceptores (*transceivers*) en la capa física del protocolo CAN.
- En 1992, Mercedes-Benz implementó el protocolo CAN en sus vehículos de la serie S. Este sistema incluía dos redes CAN: una de alta velocidad para la comunicación entre las unidades de control del motor, la transmisión y el panel de instrumentos; y otra de baja velocidad para el control del aire acondicionado y otros dispositivos internos. Esta implementación por parte de Mercedes-Benz impulsó a otros fabricantes de automóviles de lujo, como BMW y Jaguar, a adoptar redes CAN en sus modelos.
- En noviembre de 1993, el protocolo CAN fue estandarizado internacionalmente bajo la norma **ISO 11898**, que define una capa física para velocidades de transferencia de datos de hasta **1 Mbps**.

2.2. Características principales del CAN

El protocolo CAN se basa en dos capas del modelo OSI: la capa física y la capa de enlace de datos. Estas capas definen los tipos de mensajes, las reglas de acceso al bus (arbitraje) y los métodos de detección y confinamiento de fallos.[Hernández Mangas, 2019]

La velocidad del bus viene limitada por la longitud de este; es decir, si la distancia entre nodos es menor que 40 metros la velocidad de transmisión puede llegar hasta 1 Mbps. A pesar de la atenuación se llega a tener comunicación incluso a 1000 metros de distancia, pero la velocidad en este caso es de 50 Kbps. Ver figura 2.1.

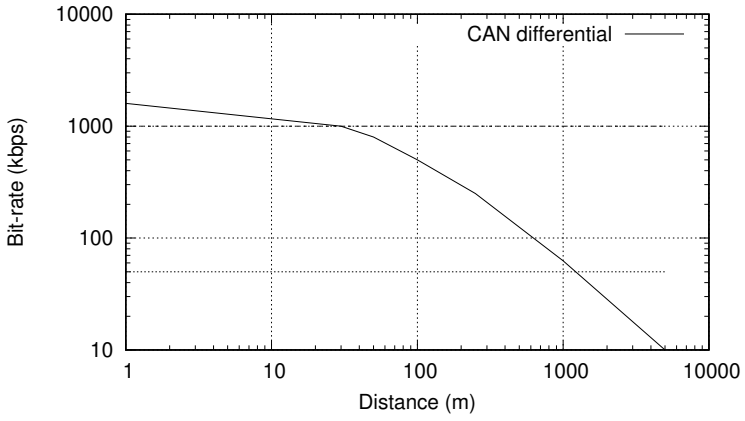


Figura 2.1: Gráfica de Longitud vs Velocidad [Hernández Mangas, 2019]

Las principales características del protocolo CAN son las siguientes: [Hernández Mangas, 2019]

- Utiliza solo dos cables, **CAN_H** y **CAN_L**, para la transmisión diferencial mediante un par trenzado. También es posible utilizar otros medios de transmisión como un hilo, radio u óptico.
- Transmite mensajes pequeños de **hasta 8 bytes** de datos, que incluyen un código CRC. El formato de los mensajes es fijo y solo se pueden transmitir cuando el bus está libre.
- **No se direccionan nodos individualmente**. En su lugar, cada mensaje tiene un identificador que especifica su contenido y los nodos interesados en ese contenido aprovechan dichos mensajes. El identificador también indica la prioridad del mensaje, siendo los identificadores numéricamente más bajos los de mayor prioridad.
- El protocolo CAN **no experimenta colisión de mensajes** a diferencia de otras tecnologías de comunicación en serie. El arbitraje de acceso al bus es no destructivo, lo que significa que cada transmisor verifica el nivel transmitido y si detecta un valor diferente, se detiene para evitar la colisión (utilizando el método **CSMA-CD**).
- El protocolo CAN dispone de un mecanismo de gestión y detección de errores, lo que garantiza una comunicación fiable y robusta.

Por último, el protocolo CAN proporciona un servicio de comunicación en el que un dispositivo puede difundir mensajes a otros sin que estos los soliciten (*Broadcasting*). Los receptores pueden aceptar o ignorar los mensajes según sus necesidades y no se requiere una confirmación del servicio. [Morales and Antonio, 2005]

2.3. Lógica del Bus

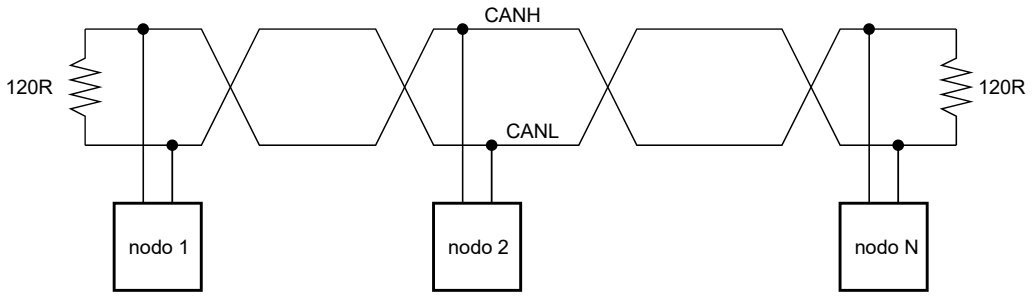


Figura 2.2: Esquema representativo de un bus CAN [Hernández Mangas, 2019]

2.3.1. Capa Física

La lógica del bus CAN se basa en el envío de señales complementarias, CAN_H y CAN_L, donde la información se codifica mediante la diferencia de tensión entre estas dos señales, como se aprecia en la figura 2.3. Esta configuración permite una mayor inmunidad al ruido externo, ya que cualquier interferencia afectará por igual a ambas señales y puede ser eliminada durante la recepción de los datos. En la figura 2.2 se muestra un ejemplo de un bus CAN en el que este debe terminar con resistencias de 120Ω para evitar las reflexiones de la señal (acoplo de impedancias correcto).

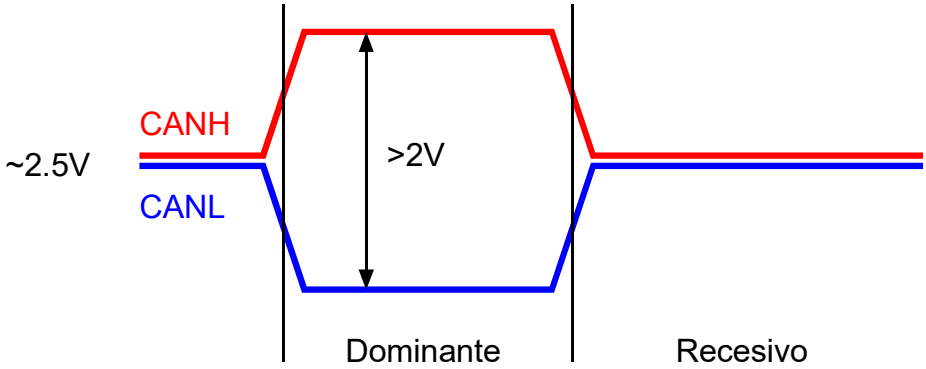


Figura 2.3: Estado dominante vs Recesivo [Hernández Mangas, 2019]

En el bus CAN, se pueden distinguir dos estados de tensión (tabla 2.1):

- **Estado Dominante**: en este estado, el nivel de tensión en el bus es más bajo, representado por una señal lógica 0. Si uno o más nodos están en estado dominante, el bus se encuentra en estado dominante. Durante la transmisión de datos, cuando un nodo envía bits en estado dominante, los demás nodos detectan esta condición y se mantienen en estado recesivo.
- **Estado Recesivo**: en este estado, el nivel de tensión en el bus es más alto, representado por una señal lógica 1. Si todos los nodos presentes en el bus están en estado recesivo, el bus se encuentra en estado recesivo. Durante este estado, los nodos pueden transmitir bits recesivos o permanecer en silencio sin interferir en la comunicación.

Nivel Lógico	Estado CAN	CANH	CANL	VcanH	VcanL
0	Dominante	VCC	GND	3,5 a 5V	0 a 1,5V
1	Recesivo	Flotante	Flotante	2,5V	2,5V

Tabla 2.1: Estados, tensiones y nivel lógico de un bus CAN [Hernández Mangas, 2019]

2.3.2. Capa de Enlace

La metodología utilizada en el bus CAN es **CSMA/CD+AMP**, que significa Acceso Múltiple con Escucha de Portadora, Detección de Colisión y Arbitraje basado en la Prioridad del Mensaje. Esta metodología permite que varios nodos compartan el medio de transmisión de manera eficiente.

En el bus CAN, el estado predeterminado es el estado recesivo, donde no hay ninguna transmisión en curso. Cuando un nodo desea transmitir una trama señala el inicio de la trama enviando un bit dominante llamado **inicio de trama** (**SOF**). Cada nodo transmisor está constantemente monitoreando el nivel del bus y lo compara con el nivel que ha transmitido. Si un nodo ha enviado un bit recesivo y detecta un bit dominante, indica que otro nodo está transmitiendo simultáneamente. En este caso, el nodo transmisor detiene inmediatamente su transmisión y se convierte en receptor de la trama. [Defaz Andrango, 2007]

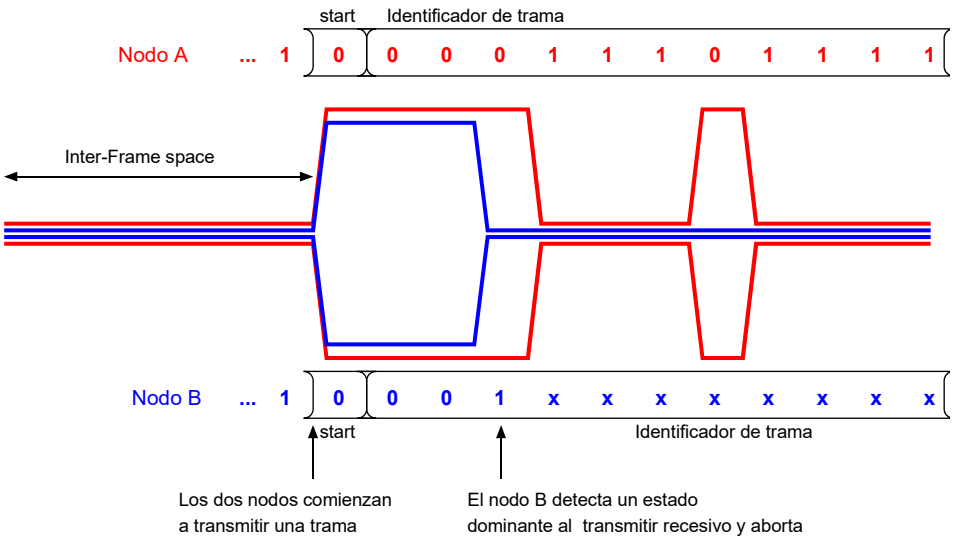


Figura 2.4: Proceso de arbitraje bus CAN [Hernández Mangas, 2019]

En la figura 2.4 se aprecia que cada mensaje en el bus tiene un identificador único asociado a él. Durante la transmisión, los bits de identificación de los mensajes determinan su prioridad. Los mensajes con identificadores de menor valor tienen mayor prioridad que los mensajes con identificadores de mayor valor.

2.4. Trama CAN

El protocolo CAN utiliza cuatro tipos de tramas [Defaz Andrango, 2007]:

- **Trama de Datos**: Se transmite desde un nodo transmisor a uno o varios nodos receptores. Contiene los datos que se desean transmitir.
- **Trama Remota**: Un nodo solicita a otro la transmisión de una trama de datos específica. Permite la comunicación bidireccional entre nodos.
- **Trama de Error**: Utilizada para señalizar errores detectados por un nodo en el bus. Esta trama destruye la defectuosa y notifica el error.
- **Trama de Sobrecarga**: Se utiliza para proporcionar un retardo adicional entre dos tramas de datos o tramas remotas.

Las tramas de datos y tramas remotas se separan de las tramas anteriores mediante un intervalo de tiempo, conocido como **IFS** o Espacio entre tramas, que consta de al menos 3 bits.

Existen dos formatos de tramas de datos compatibles pero con identificadores de diferente longitud: el **Formato Estándar** con un identificador de **11 bits** y el **Formato Extendido** con un identificador de **29 bits**.

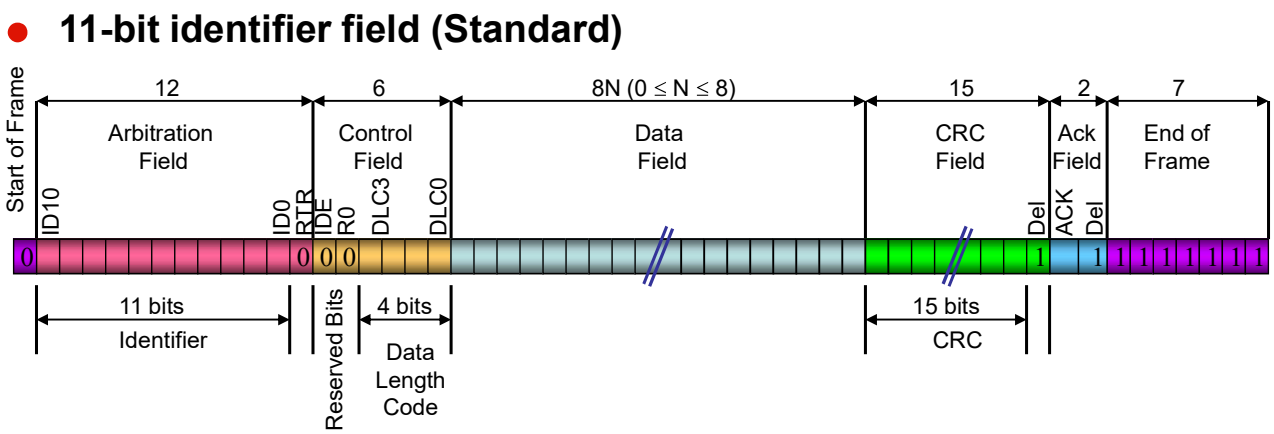


Figura 2.5: Trama estándar, identificador de 11 bits [Hernández Mangas, 2019]

2.4.1. Trama de Datos

En la figura 2.5 se muestra el formato de una trama estándar del protocolo CAN, que está compuesta por varios componentes importantes:

- **SOF** (Start Of Frame): bit dominante que indica el inicio de una trama.
- **Campo de Arbitraje**: comprende el campo **identificador** y el bit de Solicitud Remota de Transmisión (**RTR**). El identificador permite 2048 (2^{11}) o 536.870.912 (2^{29}) posibles tramas. Para priorizar la trama de datos sobre la trama remota con un mismo identificador, el bit RTR en una trama de datos se transmite en estado dominante, mientras que en una trama remota se transmite en estado recesivo.
- **Campo de Control**: el primer bit, IDE (Identifier Extension), distingue entre tramas de formato básico y extendido, siendo el formato básico el que permanece sobre el extendido. Los últimos cuatro bits son el **Código de Longitud de Datos (DLC)** estos indican el número de bytes transmitidos en el campo de datos.
- **Campo de Datos**: contiene los datos a transmitir en la trama CAN. Puede tener una longitud de 0 a 8 bytes.
- **Campo CRC** (Cyclic Redundancy Check): compuesto por una secuencia de 15 bits de verificación de redundancia cíclica y un bit recesivo de delimitación. Permite al receptor determinar si la trama recibida ha sido afectada por perturbaciones del medio.
- **Campo ACK** (Acknowledgment): cualquier receptor que haya recibido correctamente una trama lo notifica sobrescribiendo el bit recesivo transmitido con un bit dominante durante el slot del bit ACK. Al recibir un bit dominante durante este slot, el transmisor sabe que al menos uno de los nodos ha recibido correctamente la trama transmitida. [Defaz Andrango, 2007]
- **Fin de trama EOF** (End Of Frame): cada trama de datos y trama remota se delimita por una secuencia de 7 bits recesivos.

2.4.2. Trama remota

Un nodo es capaz de solicitar datos a otro nodo mediante el envío de una trama remota. Para indicar que se trata de una trama remota, el bit RTR (Remote Transmission Request) debe ser recesivo, es decir, tener un nivel lógico 1. En esta trama, no se envía ningún campo de datos, ya que su propósito es solicitar datos en lugar de transmitirlos. El campo identificador de la trama corresponde al nodo al que se solicitan los datos. De esta manera, el receptor puede identificar la solicitud y responder con la trama de datos correspondiente.[Sánchez Arias, 2018]

2.5. Temporización y Sincronización de Bit

El protocolo CAN utiliza la transmisión síncrona de datos en comparación con otros protocolos de comunicación que son asíncronos [Defaz Andrango, 2007]. Esta elección permite una mayor eficiencia en la capacidad de transmisión, pero implica un protocolo más sofisticado en términos de sincronización de bits.

En un protocolo de comunicación asíncrono, la sincronización es más sencilla, ya que se puede utilizar el bit inicial disponible en cada trama. Sin embargo, en un protocolo de comunicación síncrono, aunque se cuenta con un bit inicial disponible al inicio de la trama (sincronización dura), esto no es suficiente para mantener sincronizado el muestreo de bits en el receptor. Por lo tanto, se requiere una resincronización continua. Esta se encarga de detectar el período de reloj de la señal recibida en cada transición del bit de un estado recesivo a un estado dominante [Hernández Mangas, 2019].

La diferencia de tiempo entre las frecuencias de reloj de los nodos se compensa mediante los **segmentos de fase** situados antes y después del punto nominal de muestreo dentro de un periodo de bit, segmento de fase 1 y segmento de fase 2 respectivamente. Estos permiten realizar cambios en el punto de muestreo durante la resincronización y su variación máxima se define con **SWJ** o Anchura de Salto de Sincronización. Esta anchura puede provocar que se reduzca el segmento de fase 1, alargando el segmento 2, o bien realizando el proceso inverso. El SWJ tiene un tamaño limitado que puede ser entre 1 y el mínimo entre 4 y la longitud del segmento de fase 1.

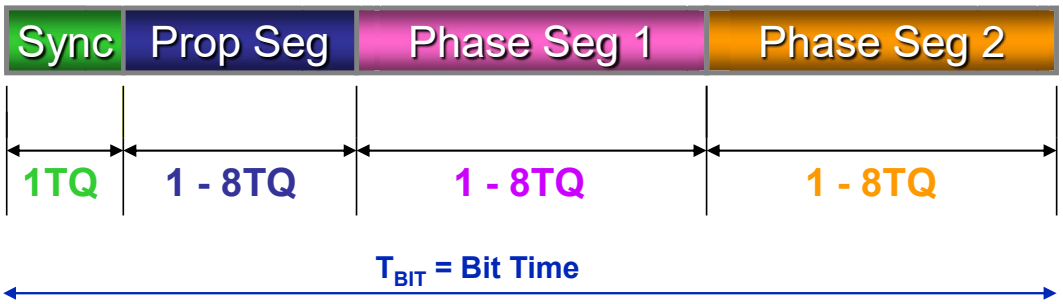


Figura 2.6: Segmentos dentro de un periodo de bit [Hernández Mangas, 2019]

El diseño del protocolo, especialmente en términos de arbitraje y acuse de recibo, requiere que la propagación de una señal desde el emisor hasta el receptor y su retorno se produzca dentro de un período de bit. Para compensar esto, se agrega un **segmento de propagación** durante la resincronización, que debe tener una longitud dos veces mayor que la suma del retardo de propagación máximo de la señal entre dos nodos a lo largo del bus y los tiempos de retardo internos de los nodos transmisor y receptor.[Defaz Andrango, 2007]

En la figura 2.6 se muestra el periodo de bit de una trama CAN dividido en los segmentos mencionados anteriormente. La longitud de estos segmentos se especifica como múltiplos de una unidad básica de tiempo T_Q (*time quantum*) derivada del periodo del oscilador.

2.6. Confinamiento de fallos

En un sistema con una topología de bus existe el riesgo de que un nodo defectuoso pueda bloquear todo el sistema. En el caso del protocolo CAN, si un nodo defectuoso genera constantemente errores, se ha incluido un mecanismo que actuaría para detectar y tomar medidas respecto a dicho nodo hasta desconectarlo del bus si es necesario. [Defaz Andrango, 2007]

Con este propósito, se han diseñado dos contadores de errores dentro de cada nodo: el **contador**

de transmisión (**TEC**) y el **contador de recepción** (**REC**). Estos contadores aumentan o disminuyen en función de si las tramas se han transmitido o recibido de manera exitosa o no.

Los contadores se incrementan de la siguiente manera:

- Si un receptor detecta un error a través del campo CRC de la trama, REC se incrementa en 1.
- Si se detecta un error después de haber enviado una trama de error, REC se incrementa en 8.
- Si un transmisor no recibe un acuse de recibo (ACK), envía una bandera de error y TEC se incrementa en 8.
- Si un transmisor detecta un error mientras envía una bandera de error, TEC se incrementa en 8.
- Si un receptor detecta un error mientras envía una bandera de error, REC se incrementa en 8.

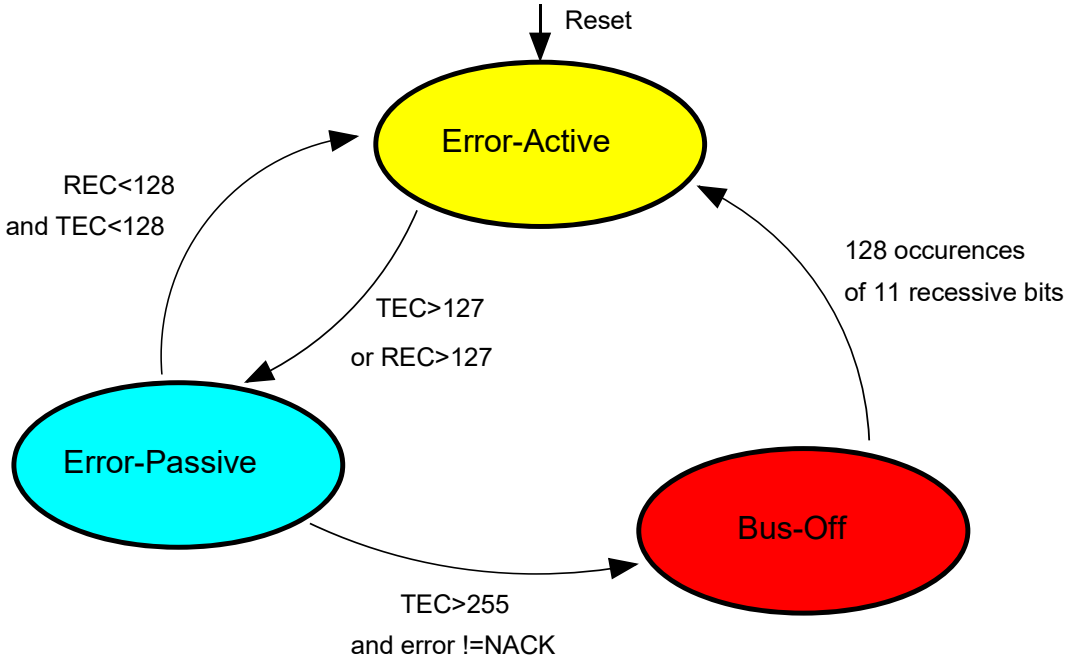


Figura 2.7: Diagrama de estados de error de un nodo.[Hernández Mangas, 2019]

En resumen, el contador de transmisión se incrementa de 8 en 8 cada vez que ocurre un error durante la transmisión y se reduce en 1 cuando hay una operación exitosa. Por otro lado, el contador de recepción aumenta de uno en uno, excepto en casos de error al transmitir tramas de sobrecarga o error, donde REC se incrementa en 8. En caso de una operación exitosa, se decrementa en 1.[Hernández Mangas, 2019]

En relación a los nodos, pueden estar en tres estados de error diferentes según los valores de los contadores TEC y REC [Defaz Andrango, 2007]:

- **Error activo**: El nodo sigue participando en la comunicación del bus y envía *flags* de error activos cuando detecta un error. Esto provoca la destrucción de la trama que se estaba transmitiendo y evita que otros nodos acepten la trama errónea.
- **Error pasivo**: El nodo ha acumulado una cantidad significativa de errores. Aunque aún puede comunicarse con otros nodos, los errores se indican mediante una bandera de error pasiva, que consta de una transmisión de 6 bits recesivos que no afectan a la comunicación del bus. Además, el nodo debe esperar 8 tiempos de bit antes de poder retransmitir la trama.

- **Bus-Off**: En este estado, al nodo no se le permite tener influencia alguna sobre el bus. No transmitirá ninguna trama, sin importar su tipo. Un nodo en estado Bus-Off puede volver al estado activo después de restablecer sus contadores y detectar 128 secuencias consecutivas de 11 bits recesivos.

Como se ilustra en la figura 2.7, un nodo cambia del estado activo al estado pasivo cuando uno de sus contadores sobrepasa el valor de 127. Para volver al estado activo, se requieren transmisiones exitosas y ambos contadores deben disminuir hasta un valor inferior a 128. Si el contador de errores de transmisión de un nodo pasivo alcanza el valor de 255, entra en el estado *Bus-Off*.

A modo de ejemplo, para ilustrar la capacidad del protocolo CAN para limitar los fallos, consideremos una aplicación en el sector de la automoción. Supongamos que un sistema funciona durante 2000 horas al año, con una tasa de transmisión de 500 kbps y un 50 % de uso del bus. Bajo estas condiciones, el sistema no será capaz de detectar un error durante un periodo de 1000 años [Hernández Mangas, 2019].

Capítulo 3

Procolo de Comunicación

Para cumplir con los objetivos mencionados anteriormente, se ha diseñado un protocolo de comunicación que sea capaz de, mediante el uso del protocolo CAN, realizar transmisiones de información de tamaño elevado entre dispositivos de poca capacidad computacional.

Este protocolo de comunicación abarca la comunicación desde un nivel de abstracción elevado, con la solicitud y recepción de información de manera transparente al usuario. Después, se desarrollará todo el proceso intermedio, en el que se realiza toda la comunicación de las tramas, el orden de envío y los códigos de operación correspondientes. Por último se hablará acerca de un nivel más bajo de abstracción, en el que se altera la composición de las tramas, además del proceso de codificación y decodificación de estas.

3.1. Protocolo a Alto Nivel

Comenzaremos con un enfoque de abstracción elevado. A partir de ahora, consideraremos tanto al cliente como al servidor como cajas negras, sin importar lo que sucede internamente en ellas.

En el diagrama de estados del servidor (ver Figura 3.2), en su estado inicial, el servidor está esperando recibir una petición del cliente (*OP_Request*). Una vez que llega la petición, se acepta o se rechaza según la decisión del servidor. En caso de rechazo, se volverá al estado inicial. Si se acepta la petición, se procede a recibir la ruta o URL del archivo solicitado(*OP_FilePath*). Una vez que se recibe la ruta completa, se busca el archivo y se envía su contenido. En caso de que no se pueda encontrar el archivo, se notificará al cliente.

Es importante señalar que en este proceso, el servidor se encarga de manejar la solicitud del cliente y proporcionar la respuesta adecuada. Así mismo, el servidor tiene la capacidad de tomar decisiones en función de la petición y su estado interno, además de notificar al cliente si se produce un error.

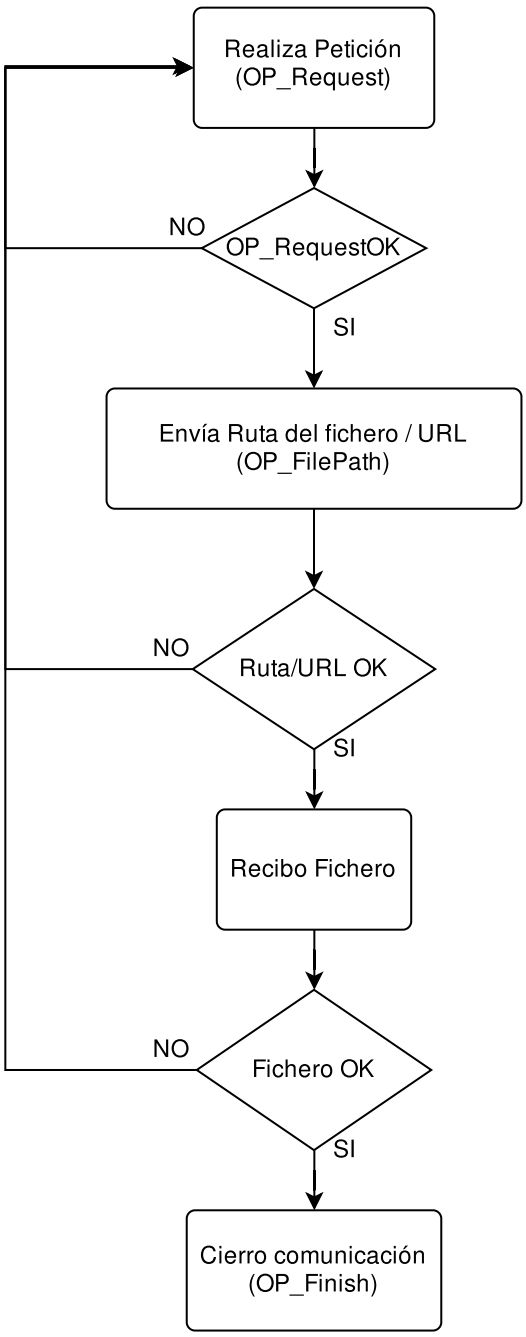


Figura 3.1: Diagrama de estados del cliente

En el lado del cliente (ver Figura 3.1), se envía una petición al servidor mediante el envío de un mensaje con el código de operación *OP_Request*. Si el servidor acepta la petición, lo notificará mediante el envío de un mensaje con código *OP_RequestOK*. En este caso, el cliente procederá a enviar la ruta del archivo solicitado (*OP_FilePath*).

Una vez que el servidor recibe la ruta completa del archivo, enviará un mensaje de confirmación. Una vez confirmada, el servidor comenzará a enviar el contenido del archivo al cliente. Este continuará recibiendo el archivo mientras comprueba que el archivo es correcto.

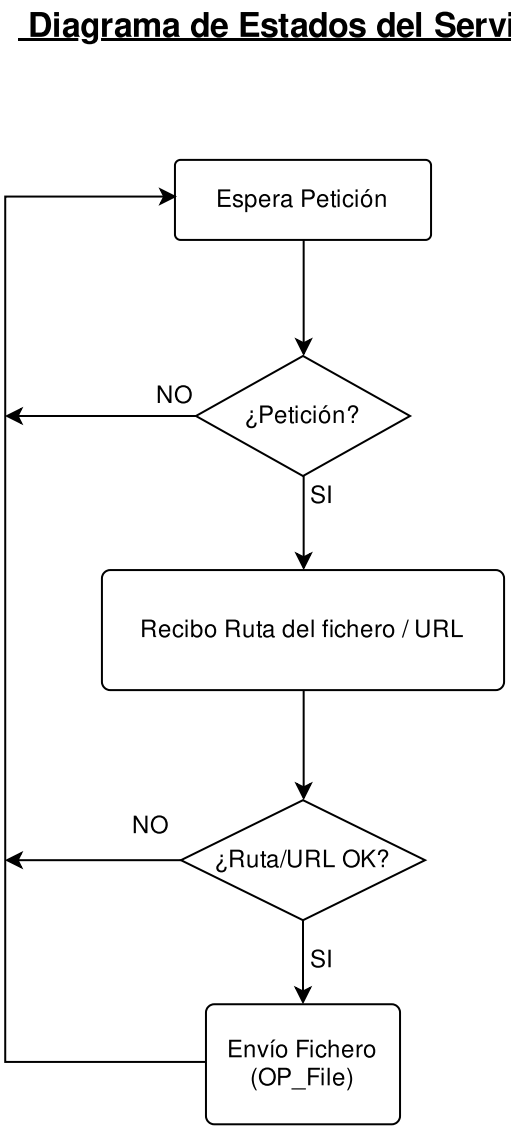


Figura 3.2: Diagrama de estados del servidor

Una vez que se ha completado la transferencia del archivo, el cliente enviará un mensaje de finalización (*OP_Finish*) al servidor, lo que indica que se ha completado la comunicación entre el cliente y el servidor.

Para terminar y a modo resumen de lo mencionado previamente, en el diagrama de flujo (figura 3.3) del proceso de comunicación entre el cliente y el servidor, se puede observar que este sigue una secuencia de mensajes bien definida. El cliente envía una petición al servidor mediante el mensaje

OP_Request y si lo acepta, enviará un mensaje de confirmación *OP_RequestOk* al cliente.

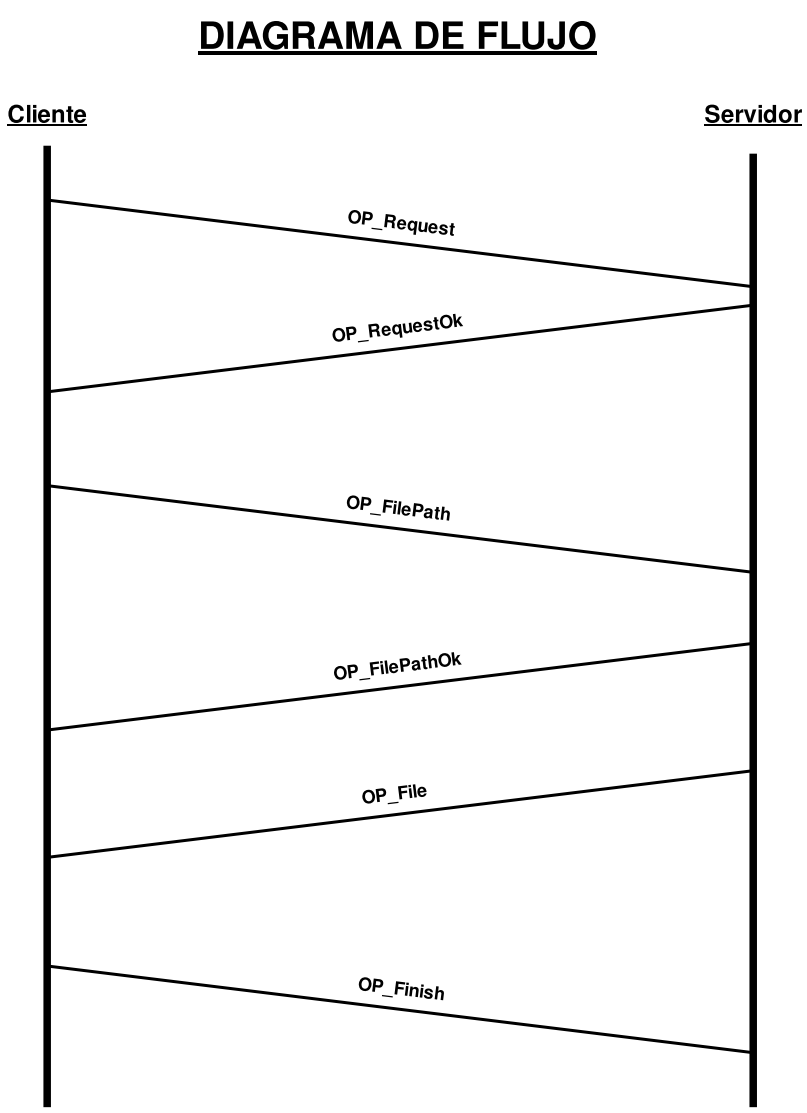


Figura 3.3: Diagrama de flujo de la comunicación a alto nivel

Cuando el cliente recibe la confirmación, enviará la ruta completa del archivo solicitado mediante el mensaje *OP_FilePath*. Al que el servidor asentirá mediante el envío de un mensaje denominado *OP_FilePathOk*. A continuación, el servidor enviará el contenido del archivo mediante el mensaje *OP_File*. El cliente lo recibirá y una vez que se haya completado la transferencia, enviará un mensaje de finalización al servidor (*OP_Finish*).

3.2. Protocolo a Medio Nivel

En este nivel de abstracción se detallará el proceso de comunicación que realizan tanto el cliente como el servidor. Primero se va a tratar el proceso de solicitud de un fichero o recurso web, este realiza una serie de comprobaciones para detectar si se ha producido algún error que en caso de que ocurran, posee mecanismos para la corrección de estos errores, hasta llegar a finalizar la comunicación si es necesario.

La parte principal de este protocolo consiste en el envío de archivos, recursos web etc., de un tamaño considerable. La forma en la que se realiza la transmisión empieza con una división del fichero completo en diferentes bloques de 2048 bytes, cada bloque se reparte en *Chunk* de 256 bytes, estos corresponden a 32 tramas completas.

3.2.1. Solicitar un fichero/URL

A continuación (ver Figura 3.4), se describe el método para enviar información de más de 8 bytes de longitud. Para ello, se ha diseñado un identificador que incluye un número de trama y un código de operación, junto con el identificador del dispositivo. La construcción del identificador se explicará más adelante, pero en este diseño se dispone de 5 bits para indicar el número de trama, por lo que se pueden enviar hasta un máximo de 32 tramas que suponen un total de $8 \times 32 = 256$ bytes para enviar la ruta deseada.

Una vez que se ha solicitado la conexión, el cliente envía la ruta de la información requerida, dividiéndola en bloques de 8 bytes para conocer el número de tramas a enviar. Se envían los primeros 8 bytes junto con el número de trama (0 en este caso) y el código de operación **OP_FilePath**. El cliente espera la respuesta **ACK** del servidor para confirmar la transmisión correcta. Después, se aumenta el número de trama y se repite el proceso si es necesario. La última trama se envía con el código de operación **OP_FilePathEnd** para indicar al servidor que es la última.

El servidor realiza las comprobaciones necesarias y envía una respuesta al cliente. Si todo está correcto, envía el OPCODE **OP_Finish** y procede a transmitir la información. Si hay algún problema con la transmisión, envía el OPCODE **OP_FilePathError**, lo que indica que el cliente debe intentar el reenvío completo de la ruta hasta tres veces antes de abortar la comunicación.

Diagrama de Estados para Solicitar un Fichero/URL

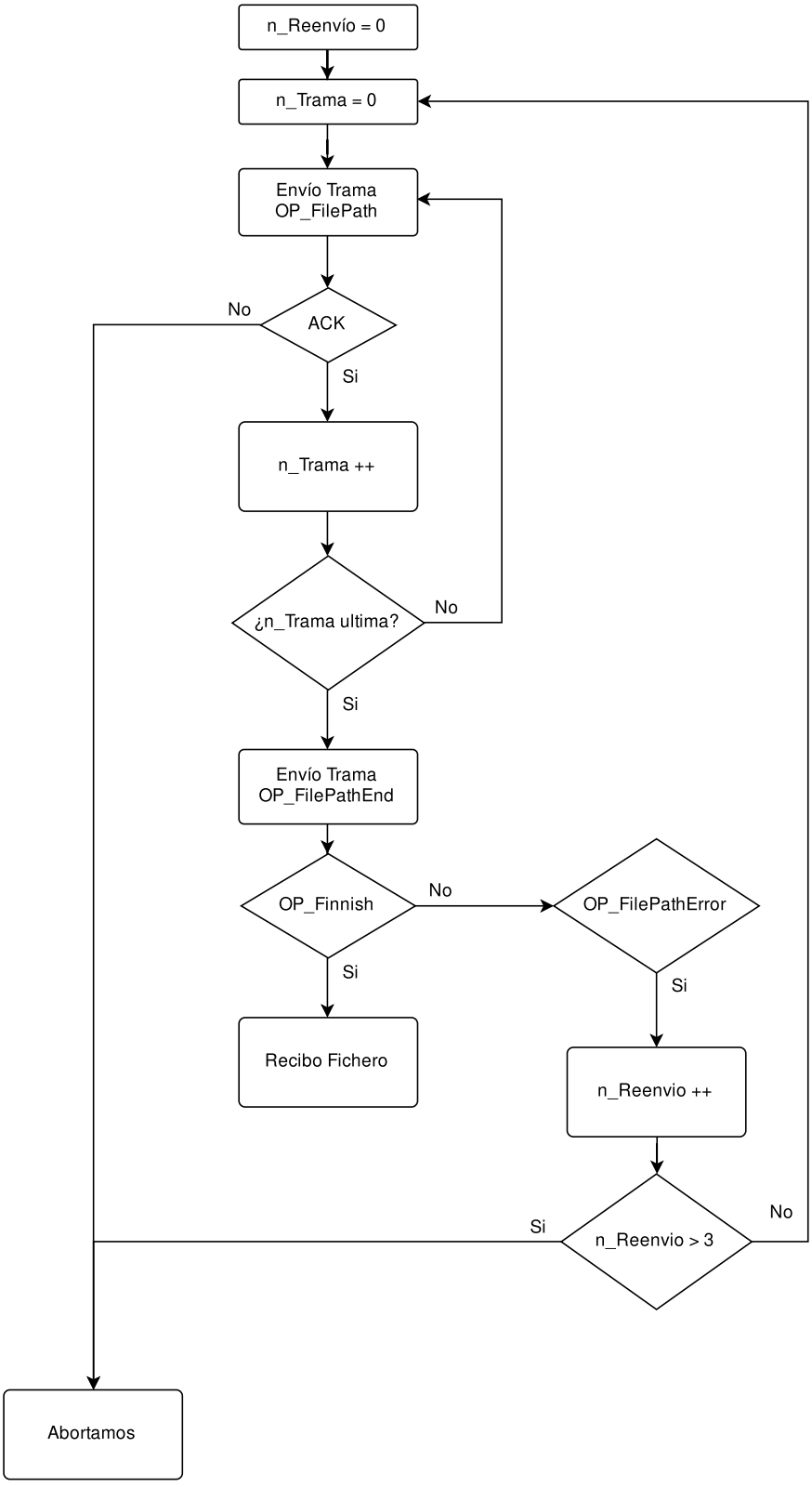


Figura 3.4: Diagrama de estados para solicitar un fichero/URL

3.2.2. Transmisión de un fichero / URL

El objetivo de este proyecto es transmitir mensajes de mayor tamaño. Por lo tanto, la forma de transmitir hasta 256 bytes que se ha descrito anteriormente resulta insuficiente. Aunque la filosofía es la misma, el protocolo consta de más comprobaciones y su complejidad es mayor.

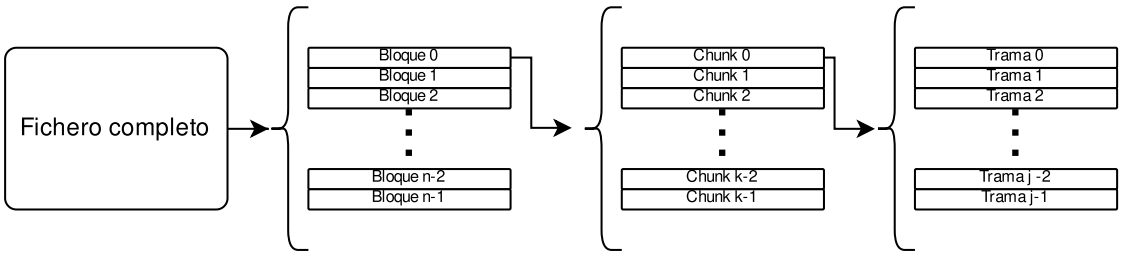


Figura 3.5: División de un fichero completo

Antes de comenzar con la transmisión, el servidor analiza la información a transmitir. Debido a que el objetivo es transmitir archivos de gran tamaño, el ESP8266 no tiene suficiente memoria para almacenar el contenido del archivo en una variable. Por este motivo, el fichero que se desee transmitir se leerá en bloques de 2048 bytes de manera recursiva hasta llegar al final del mismo. La manera de averiguar si estamos ante el último bloque es la siguiente:

- Si es un archivo dentro del sistema de ficheros, se comprueba cuánto ocupa y se divide entre el tamaño del bloque para obtener el número de bloques que se van a transmitir.
- Si se trata de un recurso web, se comprueba en cada bloque si es el final de este.

Cada bloque leído se divide entre 256, que es el número de bytes que componen un **Chunk**, como se puede ver en la figura 3.5. Un Chunk consiste en una porción de la información contenida en cada bloque, que está compuesto por un máximo de 32 tramas, ya que es el número más alto que se puede alcanzar con 5 bits para indicar el número de trama. Debido a que el tamaño de cada bloque está en base 2, se consigue que las tramas que se envíen estén siempre completas, es decir, contengan los 8 bytes de datos a excepción de la última trama, que puede tener una longitud diferente dependiendo del fichero.

Una vez calculado el número de bloques, se procede a transmitir el archivo. Como se puede ver en la figura 3.6, el servidor transmite las tramas con el código de operación **OP_File**. Antes de enviar la siguiente trama, espera a que el cliente envíe el ACK. Este proceso se hace de manera continua hasta que llega a la última trama del Chunk, que mayoritariamente será la número 31 (de 0 a 31) y se envía con el código de operación **OP_FilePartEnd**. En este momento, el receptor comprueba el número de tramas recibidas y nos envía una respuesta. Si la respuesta es con el OPCODE **OP_FilePartOk**, significa que se ha transmitido bien este bloque de 32 tramas. Pero si nos envía el código de operación **OP_FilePartError**, significa que ha habido un error y tenemos que reenviar el bloque. Este proceso es análogo a lo visto en el apartado 3.2.1.

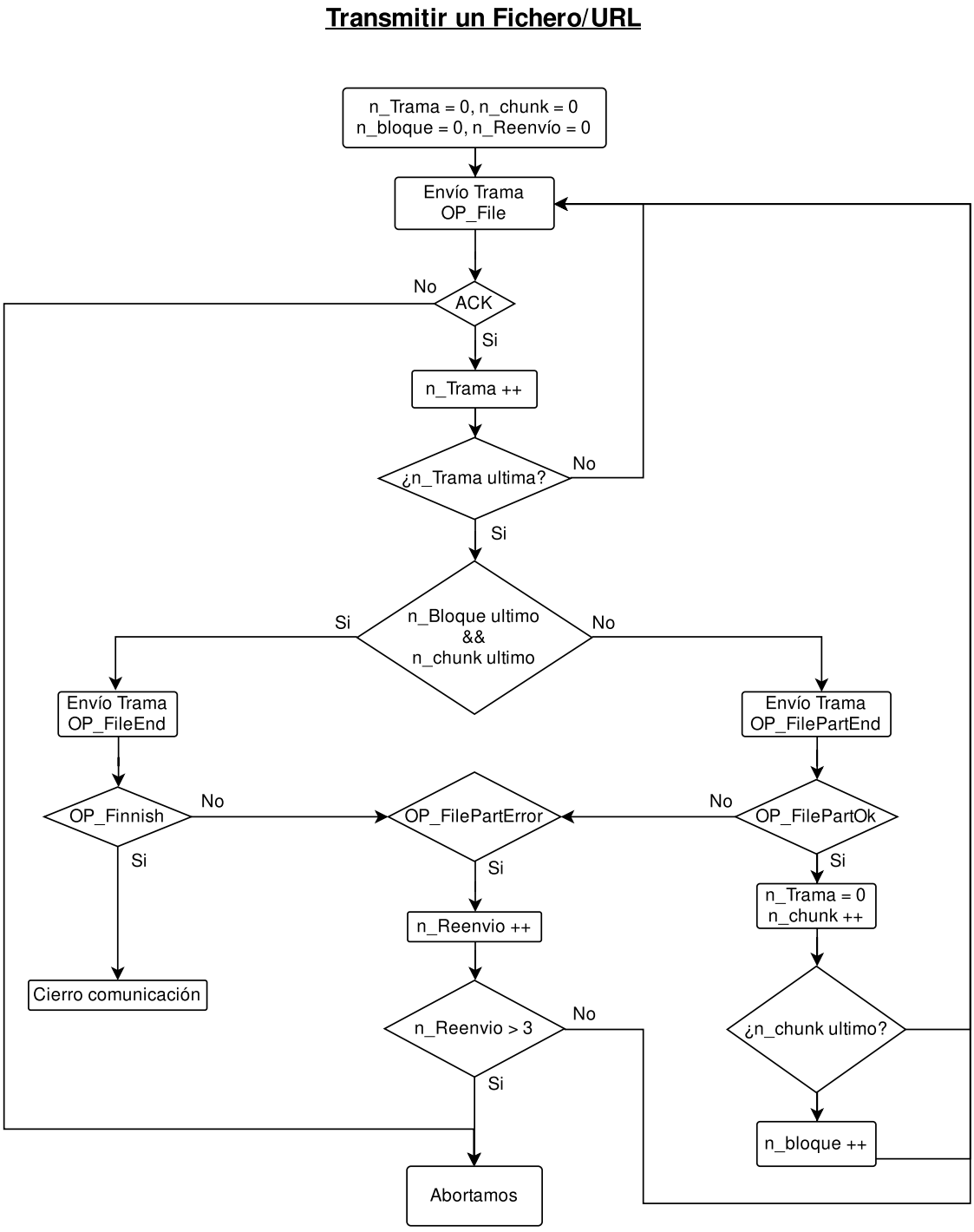


Figura 3.6: Diagrama de estados para transmitir un fichero/URL completos

Si obtenemos que la transmisión ha sido exitosa, el servidor envía el siguiente Chunk y se repite todo el proceso anteriormente descrito. Cuando se llega a la última trama del último Chunk del bloque y se ha transmitido correctamente, se lee el siguiente bloque de 2048 bytes del fichero o de la página Web y continúa transmitiéndolo de la misma manera. La comunicación finaliza cuando el servidor transmite la última trama del último Chunk, correspondiente al último bloque que la envía con el

OPCODE **OP_FileEnd** indicando al receptor que ya no hay más información que enviar. Entonces este comprueba que lo ha recibido correctamente y si ha sido exitoso envía un asentimiento con el código de operación **OP_Finnish**.

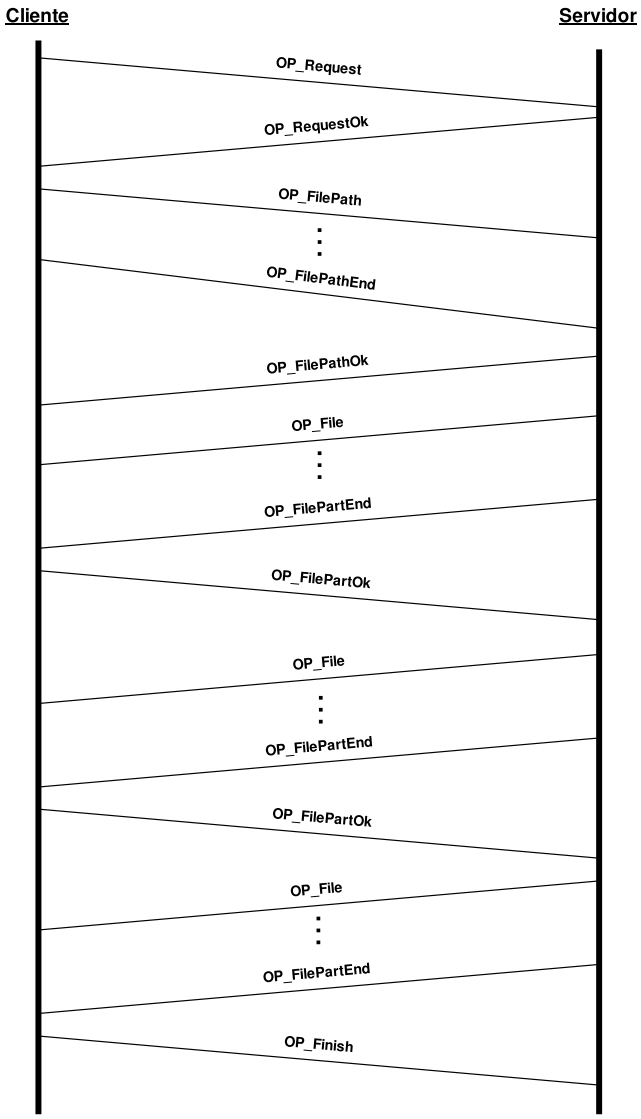


Figura 3.7: Diagrama de flujo de una comunicación a medio nivel

3.3. Protocolo a Bajo Nivel

Por último, este protocolo modifica de las tramas además de disponer de diferentes códigos de operación que se utilizan para identificar el tipo de trama, así como informar al usuario de sucedido a la hora de atender las peticiones.

Además, se describe el proceso de escritura de los campos necesarios en los registros de transmisión, así como el método inverso para la decodificación de estos campos en el receptor.

3.3.1. Cómo se identifican las tramas, OPCODE y n_trama

Las tramas CAN estándar tienen su identificador definido con 11 bits, pero también existen tramas CAN extendidas que tienen definido el identificador con 18 bits a mayores, haciendo un total de 29 bits para el identificador (ver Figura 3.8).

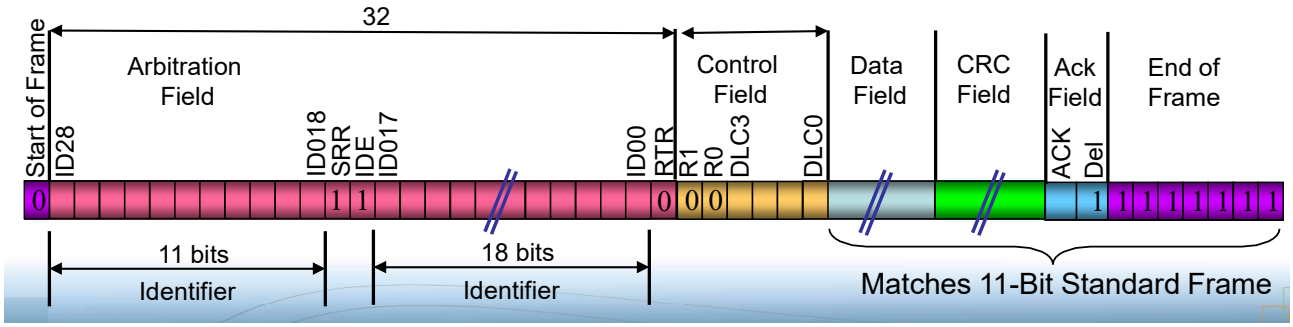


Figura 3.8: Formato de una trama CAN extendida [Hernández Mangas, 2019]

Por lo tanto, como se muestra en la tabla 3.1, vamos a aprovechar los 29 bits del identificador para enviar más información que la propia identificación, ya que utilizaremos números de identificación bajos que no requieren tantos bits para definirse. De los 29 bits reservados para el identificador, los primeros 20 se destinarán a indicar el identificador del dispositivo. Los siguientes 4 bits indicarán el código de operación u OPCODE, que tendrá 16 opciones para elegir (consulte la Tabla 3.2). Por último, los últimos 5 bits restantes se utilizarán para indicar el número de la trama enviada, que estará entre 0 y 31, ya que $2^5 = 32$.

0 — 20	21 — 24	25 — 29
Bits para el Identificador	OPCODE	nº trama

Tabla 3.1: División de los 29 bits del identificador

Implementación/Generación del ID, OPCODE y número de trama

Para empezar, el programa presenta un identificador que se utiliza para identificar cada dispositivo definido en el código como **id_propio**. El código de operación se define mediante la variable **op**, mientras que el número de trama se incluye en **trama**. Una vez que los tres parámetros están definidos, es necesario concatenarlos para obtener en una única variable el valor de estos tres campos. A partir de este momento se tratará a este conjunto como el identificador de la trama que se transmitirá en el campo *ID*. Esta variable se denomina **ID_final** y se genera mediante la siguiente línea de código:

```
ID_final = (id_propio << 9) | (op << 5) | trama;
```

En esta línea, se toman los 20 bits menos significativos de *id_propio*, lo que proporciona un rango de identificadores entre 0 y 1048575. Los 4 bits corresponden al código de operación (OPCODE), lo que permite tener 16 tipos de tramas. Por último, los 5 bits finales se utilizan para indicar las tramas desde la 0 hasta la número 31 (32 en total).

Para transmitir una trama utilizando el controlador CAN MCP2515, es necesario escribir en los registros de transmisión correspondientes [Microchip Technology Inc., 2019]:

- En el registro **TXB0SIDH** se escriben los bits más significativos del identificador final (bits 28 al 21).
- En el registro **TXB0SIDL** se escriben los bits del 20 al 18 en las 3 posiciones más altas. Estos 3 bits, junto con los escritos en el registro **TXB0SIDH**, completarían el identificador de una trama estándar. Sin embargo, si se desea transmitir tramas extendidas, es necesario incluir

los bits restantes del identificador final. Por lo tanto, en la tercera posición (**EXIDE**) de este registro, se escribe 1 para indicar que se transmitirá una trama con identificador extendido. Para finalizar con este registro, se escriben los bits más significativos del identificador extendido (bits 17 y 16) en las dos posiciones más bajas, como se muestra en la figura 3.9.

- En el registro **TXB0EID8** se escriben los bits del 15 al 8 del identificador extendido.
- En el registro **TXB0EID0** se escriben los bits menos significativos del identificador.

REGISTER 3-4: TXBnSIDL: TRANSMIT BUFFER n STANDARD IDENTIFIER REGISTER LOW (ADDRESS: 32h, 42h, 52h)

R/W-x	R/W-x	R/W-x	U-0	R/W-x	U-0	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

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

bit 7-5	**SID[2:0]:** Standard Identifier bits
bit 4	**Unimplemented:** Read as '0'
bit 3	**EXIDE:** Extended Identifier Enable bit 1 = Message will transmit Extended Identifier 0 = Message will transmit Standard Identifier
bit 2	**Unimplemented:** Read as '0'
bit 1-0	**EID[17:16]:** Extended Identifier bits

Figura 3.9: Estructura del registro TXB0SIDL [Microchip Technology Inc., 2019]

En relación al código, se procede a separar la variable *id_final* en dos partes. La primera parte contiene los 11 bits más significativos, los cuales se desplazan tres posiciones hacia la izquierda. Esto se debe a que se requiere adaptar el identificador de 29 bits a los 32 bits correspondientes a cuatro registros de 1 byte cada uno. En la segunda parte, se concatenan los 18 bits menos significativos junto con un 1 en la posición veinte, que indica una trama extendida según lo mencionado anteriormente.

Por lo tanto, en una variable auxiliar se almacenan los siguientes elementos: los 11 bits correspondientes al identificador de una trama estándar, el bit que indica una trama extendida y los 18 bits del identificador extendido. Además, se añaden los valores en las posiciones 19 y 21, aunque se escribirán en posiciones que no son relevantes. Finalmente, esta variable auxiliar se carga en los registros mencionados, byte por byte.

```
        // Adapto los identificadores de la trama
    id1 = (Msg->ID << 3) & 0b11111111111000000000000000000000;
    id2 = (Msg->ID & 0b11111111111111111) | (0b010 << 18); // Nos aseguramos IDE
        =1

    int idf = id1 | id2;

    // Guardo los diferentes bits del ID en sus correspondientes registros
    MCP2515_Write(MCP_TXB0SIDH, (char)((idf & 0xFF000000) >> 24));
    MCP2515_Write(MCP_TXB0SIDL, (char)((idf & 0x00FF0000) >> 16));
    MCP2515_Write(MCP_TXB0EID8, (char)((idf & 0x0000FF00) >> 8));
    MCP2515_Write(MCP_TXB0EID0, (char)((idf & 0x000000FF)));
```

Códigos de Operación

Los códigos de operación se pueden clasificar en diferentes tipos según su función:

- **Códigos de petición**: estos códigos se utilizan para enviar solicitudes y sus respectivas respuestas. Incluyen los códigos **OP_Request** y **OP_RequestOK**.
- **Códigos relacionados con la solicitud del fichero**: estos códigos se utilizan para enviar la ruta o URL del archivo solicitado. Incluyen los códigos **OP_FilePath**, **OP_FilePathEnd** y **OP_FilePathError**.
- **Códigos relacionados con el envío del fichero**: estos códigos se utilizan para enviar el archivo solicitado. Incluyen los códigos **OP_File**, **OP_NoFile**, **OP_FilePartEnd**, **OP_FilePartOk** y **OP_FilePartError**.
- **Códigos de información del servidor**: estos códigos se utilizan para enviar información del servidor, su estado actual o si ha ocurrido alguna incidencia. Incluyen los códigos **OP_OtherRequest**, **OP_NoInternet** y **OP_NoCapacity**.
- **Códigos de asentimiento**: estos códigos se utilizan para confirmar o finalizar una operación. Incluyen los códigos **OP_ACK** y **OP_Finnish**.

Bits (Hexadecimal)	OPCODE	Significado
0x0	OP_REQUEST	El cliente realiza una petición al servidor
0x1	OP_RequestOk	Acepta conexión
0x2	OP_FilePath	Se envia el path del fichero
0x3	OP_FilePathEnd	Se termina de enviar el path del fichero
0x4	OP_FilePathError	Se detecta un error en las tramas filepath
0x5	OP_NoFile	Indica que no existe el fichero
0x6	OP_File	Envia el fichero
0x7	OP_FilePartEnd	Se ha enviado una parte del fichero
0x8	OP_FilePartOK	La parte enviada es correcta
0x9	OP_FilePartError	Hay un error en la parte enviada
0xA	OP_FileEnd	Se ha enviado todo el fichero
0xB	OP_ACK	Confirma la recepción de la información
0xC	OP_Finish	Se finaliza la comunicación
0xD	OP_OtherRequest	Recibe otra solicitud
0xE	OP_NoInternet	Indica que no pudo acceder a internet
0xF	OP_NoCapacity	Indica que el fichero es demasiado grande para almacenarlo

Tabla 3.2: OPCODES disponibles

3.3.2. Decodificación de una trama recibida

Ahora en el receptor, el objetivo es descomponer la trama recibida, tanto los datos del archivo recibido como los 29 bits del identificador para ubicar la trama en la máquina de estados.

Las tramas recibidas se clasifican en función de si son tramas extendidas o no, o de si son tramas RTR (Remote Transmission Request) o no. Puede haber varias combinaciones posibles de estas dos variables. En nuestro sistema, esta clasificación se implementa en las siguientes líneas de código:

```
        if (((status >> 4) & 0x01) == 1)
        type = 1; // Extended identifier
    else
        type = 0; // Standard identifier
    if (((status >> 3) & 0x01) == 1)
        remote = 1; // Standar identifier Remote frame
    else
        remote = 0; // Extended identifier Remote Data frame
```

Una vez clasificada la trama, es necesario determinar en qué *buffer* de recepción se ha guardado la trama recibida considerando que hay dos *buffers* disponibles en el MCP2515 [Microchip Technology Inc., 2019]. Los registros de recepción son similares a los mencionados para la transmisión y se distribuyen de la siguiente manera:

- El contenido del registro **MCP_RXBnSIDH**, que contiene los bits más significativos del identificador, se almacena en la variable **SIDH**.
- El contenido del registro **MCP_RXBnSIDL**, que incluye los 3 bits menos significativos del identificador estándar, el indicador de trama extendida y los bits 18 y 17 del identificador extendido, se almacena en la variable **SIDL**.
- El contenido del registro **MCP_RXBnEID8**, que incluye los bits 15-8 del identificador extendido, se almacena en la variable **EID8**. El contenido del registro **MCP_RXBnEID0**, que incluye los bits 7-0 del identificador extendido, se almacena en la variable **EID0**. El contenido de los 4 primeros bits del registro **MCP_RXBnDLC**, que indican el tamaño de los datos enviados en bytes (puede ser de 0 a 8 bytes), se almacena en la variable **DLC**.

Una vez leídos los registros, se decodifica la información de manera inversa a la transmisión. Para obtener el número de la trama, se toman los 5 bits menos significativos del identificador completo que se encuentran en la variable **EID0**. Los datos recibidos se leen byte a byte de manera secuencial desde los registros **MCP_RXBnD0** hasta **MCP_RXBnD8** y se almacenan en la variable **txt** con un desplazamiento (*offset*) correspondiente al número de la trama.

```
        i = MCP2515_ReadStatus();
        if (i & 0x01) // Buffer 0
    {

        SIDH = MCP2515_Read(MCP_RXB0SIDH);      // Standard identifier high
        SIDL = MCP2515_Read(MCP_RXB0SIDL);      // Standard identifier low
        EID8 = MCP2515_Read(MCP_RXB0EID8);      // Extended identifier high
        EID0 = MCP2515_Read(MCP_RXB0EID0);      // Extended identifier low
        DLC = MCP2515_Read(MCP_RXB0DLC) & 0xF; // Data length code

        n_trama = EID0 & 0b11111;

        for (i = 0; i < DLC; i++)
            txt[i + (8 * n_trama)] = MCP2515_Read(MCP_RXB0D0 + i);
    }
    else // Buffer 1
    {
        SIDH = MCP2515_Read(MCP_RXB1SIDH);      // Standard identifier high
        SIDL = MCP2515_Read(MCP_RXB1SIDL);      // Standard identifier low
        EID8 = MCP2515_Read(MCP_RXB1EID8);      // Extended identifier high
        EID0 = MCP2515_Read(MCP_RXB1EID0);      // Extended identifier low
        DLC = MCP2515_Read(MCP_RXB1DLC) & 0xF; // Data length code

        n_trama = EID0 & 0b11111;

        for (i = 0; i < DLC; i++)
            txt[i + (8 * n_trama)] = MCP2515_Read(MCP_RXB1D0 + i);
    }

    OPCODE = ((EID8 & 0b1) << 3) | (EID0 >> 5);
    id_ajeno = (EID8 >> 1) | ((SIDL & 0b11) << 7) | ((SIDL >> 5) << 9) | (SIDH <<
        12);
```

Para obtener el código de operación y el ID del dispositivo, se aplican máscaras y desplazamientos a los bits relevantes. De esta manera, al concatenarlos, se obtiene de manera correcta el identificador y el código de operación de la trama enviada por otro dispositivo.

3.4. Gestión de Errores

La gestión de errores en un protocolo de comunicación es una parte fundamental y crucial para garantizar una transmisión fiable y precisa de la información. Si bien el envío de datos es importante, el control y el manejo adecuado de los errores que pueden ocurrir en el sistema son igualmente esenciales.

En cualquier entorno de comunicación, pueden surgir diversos tipos de errores, como la corrupción de datos, la pérdida de paquetes y la interferencia electromagnética entre otros. Estos errores pueden tener un impacto significativo en la integridad de los datos transmitidos y en la fiabilidad del sistema en general.

Por lo tanto, es fundamental implementar estrategias y técnicas de gestión de errores para detectar, corregir y recuperarse de manera eficiente de los posibles errores que se presenten durante la comunicación. Esto implica el uso de algoritmos de detección y corrección de errores, como sumas de verificación, códigos de redundancia cíclica (CRC), mecanismos de retransmisión, control de flujo y control de congestión entre otros.

3.4.1. Comprobación de la total recepción de las tramas

La comprobación de que se han recibido todas las tramas enviadas se realizará mediante la suma del número de las tramas recibidas. Este método se basa en la anécdota del cálculo de la suma de los 100 primeros números que **Gauss** realizó en el siglo `XVIII`, cuando solo tenía 10 años de edad. A partir de este cálculo, se obtuvo la expresión:

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2} \text{[Velázquez, 2015]} \tag{3.1}$$

La expresión 3.1 será implementada en el código para verificar que se han recibido todas las tramas enviadas por el transmisor. Para determinar si se ha producido alguna perdida de tramas durante la transmisión, se compara el resultado de la suma de los números de trama recibido con el valor calculado a través de la formula. La forma de implementarlo se dividirá en dos partes:

- Durante el proceso de recepción de tramas, se realiza la suma de cada número de trama recibida para mantener un registro de las tramas recibidas. Además, se almacena el número de trama más alto encontrado hasta el momento. Esta técnica resulta especialmente útil cuando el transmisor envía las tramas en orden inverso, ya que el número de trama más alto determina el valor total de la suma. En caso de no almacenar el máximo, si las tramas se envía en orden inverso, el último número de trama sería 0, por lo que la suma resultante también sería 0.

```
// Sumamos el n_trama de las tramas recibidas para hacer checksum con las tramas
    .
checkFrame += n_trama;
        if (n_trama > n_trama_max)
            n_trama_max = n_trama;
```

- Si, por ejemplo, se recibe el final de un bloque o *chunk* con el código de operación *OP_FilePartEnd*, se llama a la función **checkTramas()** pasando como parámetros el número de trama más alto y la suma de los números de trama realizada por el receptor. Esta función realiza la suma de la expresión 3.1 para n = finalFrame y luego compara el resultado con la suma calculada por el receptor. Si ambas sumas coinciden, indica que se han recibido todas las tramas enviadas por el transmisor y la función devuelve 0. En cambio, si las sumas no coinciden, se devuelve -1, lo cual indica que se ha producido un error.

```
int checkTramas(int finalFrame, int sumFrames)
{
        // Sumatorio del tramas cogiendo la ultima recibida.
    int sumTramasT = (finalFrame * (finalFrame + 1)) / 2;
```

```
    if (sumTramasT == sumFrames)
        return 0;
    else
        return -1;
}
```

3.4.2. Enviar información del estado actual

El servidor ha sido programado para informar al cliente sobre cualquier problema que pueda surgir durante la solicitud de datos. Para notificar estos problemas, el servidor envía una trama con un código de operación específico para cada tipo de error. Los diferentes tipos de error y sus códigos de operación correspondientes son los siguientes:

- Si se produce un fallo de conexión a Internet, ya sea debido a la falta de conectividad o a la indisponibilidad de la página web solicitada, el servidor envía al cliente una trama con el código de operación **OP_NoInternet**.
- En caso de que el archivo solicitado no exista en el sistema de ficheros del ESP8266, el servidor envía una trama con el código de operación **OP_NoFile**.
- Si se intenta descargar un archivo cuyo tamaño es mayor que el espacio disponible en el ESP8266, el servidor cancela la descarga y envía una trama al cliente con el código de operación **OP_NoCapacity**.

Tanto el servidor como el cliente tienen implementado un control de errores para detectar y manejar posibles problemas durante el envío de las tramas. En el caso específico de la transmisión de la ruta de un archivo, si el cliente completa el envío de la ruta y el servidor detecta algún error en alguna trama, lo comunica al cliente utilizando el código de operación **OP_FilePathError**.

Ante esta situación, el cliente tiene la capacidad de reenviar el *Chunk* de datos corrupto hasta un máximo de 3 intentos antes de abortar la comunicación. Esta verificación y retransmisión se realiza cada vez que se transmiten múltiples tramas, tanto en el lado del cliente como en el lado del servidor.

Este control de errores garantiza que se realice una verificación exhaustiva de la integridad de los datos transmitidos y se brinda la oportunidad de corregir posibles errores en la comunicación antes de considerarla como fallida. Esto aumenta la fiabilidad y la robustez del protocolo de comunicación utilizado entre el servidor y el cliente.

3.4.3. Uso de temporizadores para evitar puntos de bloqueo

Uno de los problemas que puede surgir en el protocolo de comunicación es la presencia de problemas en el medio físico. En este caso, sería el bus utilizado para la comunicación entre los dispositivos. Estos problemas pueden ocasionar una pérdida de conexión, lo que impide que, tanto el cliente como el servidor, sean capaces de resolverlo mediante el confinamiento de fallos del protocolo CAN [Hernández Mangas, 2019].

Para hacer frente a esta situación, se emplean temporizadores como mecanismo de control. Por ejemplo, cuando se envía una trama y se espera una respuesta, se establece un límite de tiempo de espera de 900 ms. Si la respuesta no se recibe dentro de ese período de tiempo, se aborta la comunicación, ya que se asume que ha ocurrido un problema en el medio físico.

Además, otro escenario que se considera es la acumulación de tramas en el *buffer* de transmisión. Si las tramas permanecen en el *buffer* durante más de 900 ms, también se aborta la comunicación, ya que indica un posible problema en el medio físico de transporte.

Estos mecanismos de temporización y finalización de la comunicación permiten detectar y gestionar de manera efectiva los problemas relacionados con el medio físico de transporte, garantizando una comunicación fiable y evitando situaciones en las que los dispositivos queden "*a ciegas*"sin poder recuperarse automáticamente mediante las técnicas de corrección de errores internas.

3.5. Rendimiento

El rendimiento del sistema de comunicación se evalúa en función de diversos parámetros, como la velocidad de transferencia, que en este caso es de **125.000 bits por segundo**, el tiempo de respuesta y la eficiencia en la transmisión de datos. En nuestro caso, hemos realizado mediciones para evaluar el tiempo de transferencia al solicitar el fichero **PCB_top.pdf** del sistema de ficheros del ESP8266.

Durante las pruebas, sabiendo que el tamaño del fichero es de **92371 bytes**. Para medir el tiempo de transferencia, se registró el intervalo desde el inicio de la solicitud del fichero hasta la finalización de la comunicación, obteniendo un valor de **22 segundos**.

$$Tamaño\ del\ fichero = \ 92371\ Bytes\ = \ 738968\ bits$$

En el caso de nuestro sistema de comunicación, cada trama tiene la capacidad de contener hasta 8 bloques de 1 byte cada uno, lo que equivale a un total de 64 bits por trama. Ahora, para determinar la cantidad mínima de tramas necesarias para transmitir todo el fichero de 738968 bits, realizamos el siguiente cálculo:

$$\frac{738968\ bits}{64\ bits/trama}\ =\ 11546{,}375\ tramas$$

Se utilizan tramas con identificador extendido, lo que implica que cada trama tiene una longitud de 65 bits para datos de control, como se puede ver en la tabla 3.3. Además de los 64 bits de datos a transmitir. Por lo tanto, cada trama consta de un total de 129 bits.

Campo	Bits	Propósito
Start of frame	1	Debe ser 0 (dominante)
Identificador	29	Identificador del mensaje
Remote Transmision Request	1	0: data frame, 1: RTR frame
ID extension	1	Debe ser O
Reserved	1	Debe ser O
Data Length Code	4	Nº de bytes de datos (0-8)
Data Field	DLC*8	Datos
CRC	15	Comprobación de errores
CRC delimiter	1	Debe ser 1
ACK	1	Los receptores ponen este bit en 0
ACK delimiter	1	Deber ser 1
End of Frame	7	Deben ser 1111111
Inter-Frame Spacing	3	Deben ser 111

Tabla 3.3: Bits que componen una trama CAN extendida [Hernández Mangas, 2019]

Para determinar la cantidad total de bits que se necesitan enviar, simplemente multiplicamos el número de tramas requeridas por el número de bits por trama:

$$bits_{FILE}\ =\ 11546{,}375\ tramas \times\ 129\ bits/trama\ =\ 1489482{,}375\ bits$$

Teniendo en cuenta la velocidad de transmisión de las tramas CAN que, como se ha mencionado anteriormente, es de 125.000bps, podemos calcular el tiempo total necesario para transmitir todas las tramas. Para el fichero, se requiere transmitir un total de 1.489.482.375 bits, por lo que el tiempo necesario se calcula de la siguiente manera:

$$tiempo_{FILE} \;=\; \frac{1489482{,}375\ bits}{125000\ bps} \;=\; 11,91\ segundos$$

Esto significa que se necesitan aproximadamente 12 segundos para transmitir todas las tramas del fichero.

Además, por cada trama recibida, el cliente envía una trama de asentimiento (ACK) que tiene una longitud de datos de 0. Por lo tanto, el número de bits necesarios para transmitir las tramas ACK se calcula de la siguiente manera:

$$bits_{ACK} \;=\; 11546{,}375\ tramas \;\times 65\ bits/trama \;=\; 750514{,}375\ bits$$

El tiempo necesario para transmitir las tramas ACK se calcula como:

$$tiempo_{ACK} \;=\; \frac{750514{,}375\ bits}{125000\ bps} \;=\; 6\ segundos$$

Por lo tanto, el tiempo total estimado para transmitir tanto el fichero como las tramas ACK es de aproximadamente 18 segundos. Cabe destacar que este cálculo no incluye el tiempo de procesamiento de las tramas ni el tiempo desde que se envía la solicitud hasta que se recibe el primer bloque del fichero, por lo que el tiempo real puede ser mayor.

En el caso de la prueba real, se midió un tiempo de 22 segundos para la transmisión del mismo fichero. Dado que este tiempo incluye el procesamiento y otros posibles retrasos, podemos afirmar que el tiempo teórico calculado se ajusta razonablemente bien al tiempo medido en la prueba real.

Capítulo 4

Ejemplo de Aplicación

En el desarrollo de este proyecto, se ha creado un prototipo que actúa como una interfaz entre Internet y el protocolo CAN, permitiendo dotar a los dispositivos de conexión a Internet a través del uso de este protocolo. Este sistema consta de una parte *hardware*, que se detallará en el apartado 4.1, donde se ha diseñado una PCB que tiene como componente central un **ESP8266** que utiliza el entorno de desarrollo **Arduino**.

La placa también incorpora los componentes necesarios para la transmisión y recepción de señales CAN, como el **controlador CAN MCP2515** y el **transceptor CAN MCP2551**, ambos de *Microchip Technology Inc*[Microchip Technology Inc., 2019]. Estos dispositivos se encargan de gestionar las solicitudes realizadas por el microcontrolador a través del protocolo **SPI**, así como de informar al microcontrolador sobre los eventos que ocurren en el bus. Además, se ha incluido una interfaz UART en el prototipo para permitir la comunicación serie con un PC mediante el puerto Micro-USB ubicado en un lateral de la placa. Esto facilita el proceso de programación y depuración del ESP12-F.

Una vez definido el *hardware* utilizado, es necesario programar el *software* (sección 4.2) que cumpla con los requisitos del proyecto. Este programa permite establecer la comunicación con el bus CAN a través del transceptor MCP2551, así como procesar y ejecutar diversas acciones según las necesidades específicas en cada momento, que abarcan desde recibir solicitudes de otros dispositivos hasta descargar y enviar código HTML de páginas web, con el objetivo de superar las limitaciones inherentes a dicho protocolo.

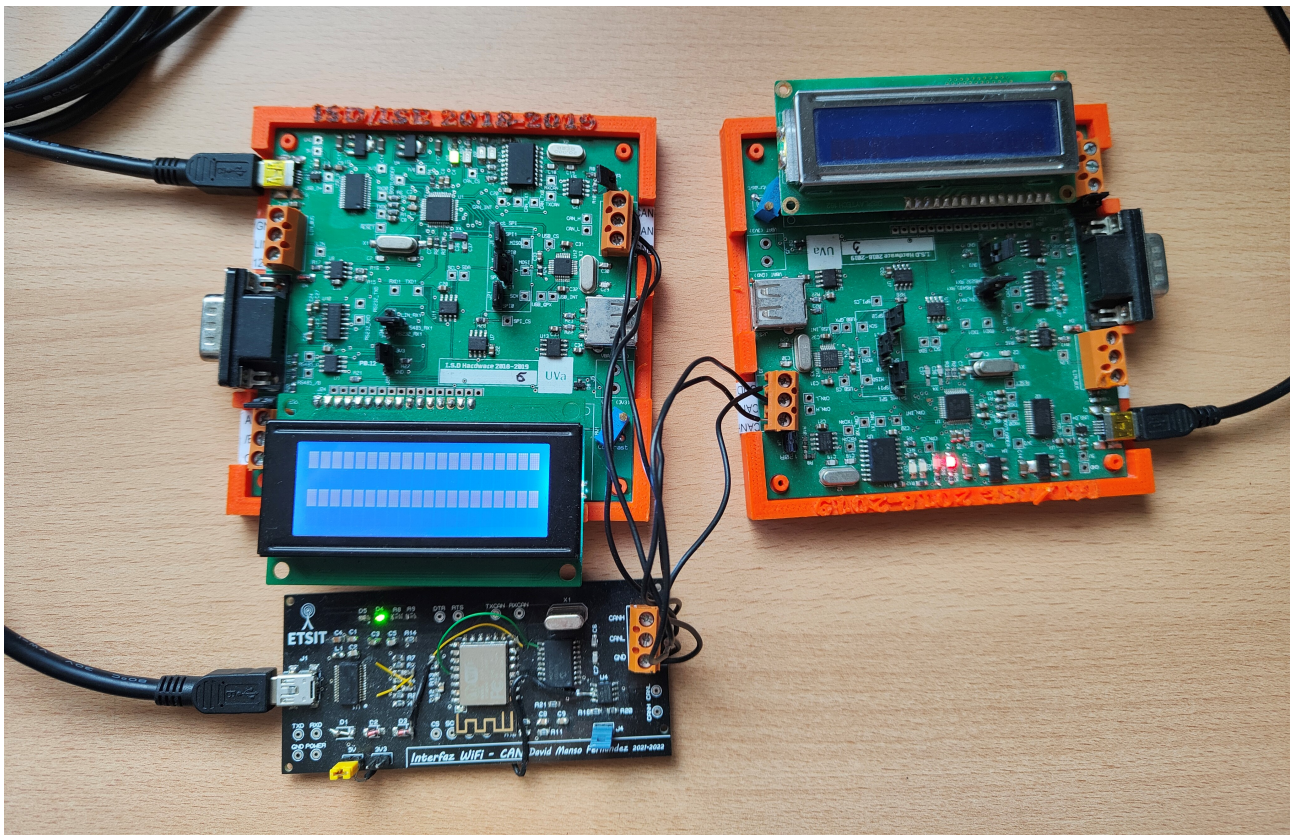


Figura 4.1: Ejemplo de uso

Finalmente, se ha diseñado una interfaz de usuario que proporciona una representación visual de los eventos que ocurren tanto en nuestro sistema basado en el ESP8266 como en el dispositivo auxiliar de prueba. Este dispositivo auxiliar consiste en una PCB que utiliza un **microcontrolador LPC2103** y cuenta con los mismos componentes *hardware* que nuestro prototipo. La interfaz de usuario permite obtener una retroalimentación visual de las acciones que se realizan, tanto en el sistema propio como en el dispositivo de prueba, que actúa exclusivamente como cliente y solicita diferentes archivos o servicios. En la figura 4.1 se muestra la disposición de todos los sistemas que están conectados al bus CAN.

4.1. Hardware

Para el diseño del sistema *hardware* se utilizó como punto de partida un prototipo desarrollado por Jesús M. Hernández Mangas (figura 4.2). El prototipo original constaba de dos placas independientes que se conectaban mediante pines y se complementaba con el componente CH340G. El CH340G es un dispositivo integrado que incluye una UART, un cristal generador de la señal de reloj y un puerto USB-A macho para la conexión al PC. A mayores, aporta las señales necesarias para el funcionamiento completo del sistema, como las de alimentación de **3,3V** y **5V**, las señales de control **DTR** y **RTS**, las señales de datos **TXD** y **RXD**, y el pin de tierra **GND**.

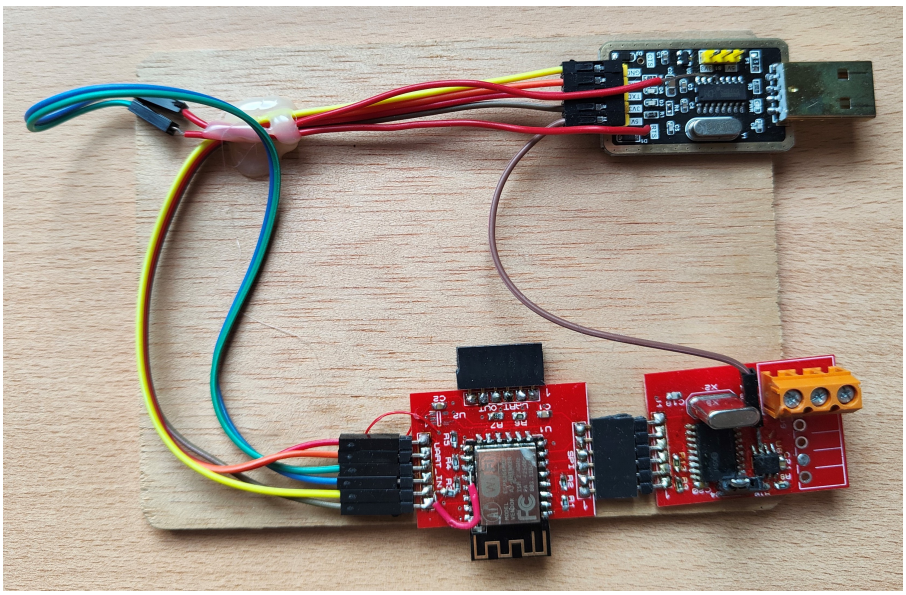


Figura 4.2: Prototipo inicial

En el diseño original, una de las placas albergaba el ESP8266, mientras que la otra se encargaba de todo lo relacionado con la transmisión CAN. Sin embargo, este prototipo presentaba algunos problemas, especialmente en cuanto a su robustez física, ya que cualquier movimiento mínimo podía causar funcionamientos erróneos. Además, estaba diseñado para funcionar con diversos sistemas, lo que no era óptimo para las necesidades específicas de este proyecto.

Por tanto, se tomó la decisión de integrar todos los componentes en una única placa y diseñarla de manera específica, al igual que se buscó mejorar la robustez y la estabilidad del sistema, para cumplir con los requisitos y objetivos de este proyecto en particular. La placa diseñada consta de tres partes distintas que desempeñan funciones específicas en el sistema:

- **Comunicación con PC externo**: esta parte está encargada de la comunicación entre la placa y un PC mediante el puerto USB mini-B hembra y una UART. El puerto permite la conexión física con el PC a través de un cable, mientras que la UART proporciona la interfaz de comunicación serie entre ambos dispositivos.
- **Núcleo central**: el núcleo central del sistema es el ESP8266, que actúa como la interfaz principal entre Internet y el protocolo CAN.
- **Comunicación CAN**: esta parte del diseño se encarga de la comunicación con el bus. Incorpora el controlador MCP2515 y el transceptor MCP2551. Además, incluye un cristal de cuarzo de 20MHz para proporcionar una señal de reloj precisa al controlador MCP2515 y un puerto TBLOCK-l3 que se utiliza para la conexión física con el bus CAN, permitiendo la transmisión y recepción de datos.

En las figuras 4.3 y 4.4 se muestra el resultado final del prototipo diseñado y la suma de los componentes que se montan en esta PCB es de 14,46€ a día 23 de mayo de 2023 como se puede comprobar en el anexo 5.3.

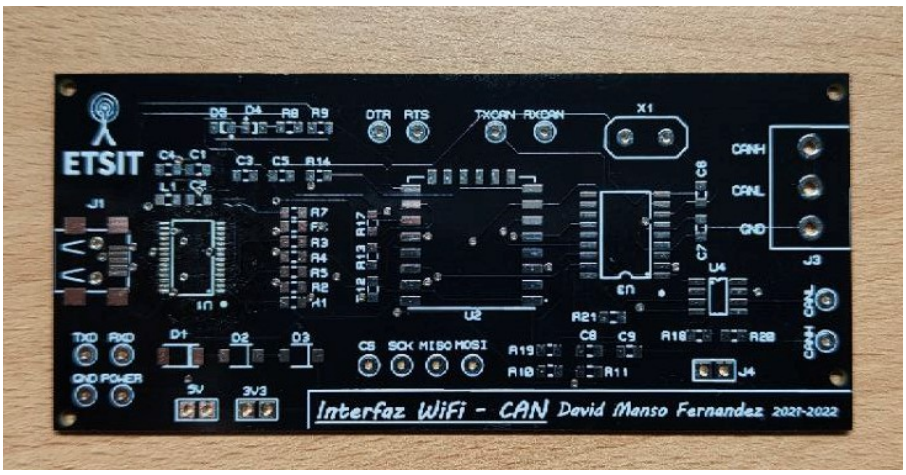


Figura 4.3: Cara A de la PCB fabricada

Figura 4.4: Cara B de la PCB fabricada

4.1.1. Comunicación con PC externo

La comunicación entre un PC externo y el ESP8266 se establece mediante la utilización de la UART y el puerto USB mini-B. El puerto USB (J1) permite la conexión de un cable proveniente del PC que contiene el código fuente del programa que se desea cargar en el sistema. Además de programar el ESP8266, el PC externo también suministra la alimentación al sistema. Se tiene previsto que en futuras iteraciones de este sistema, pueda funcionar de manera independiente a un ordenador externo utilizando una batería como fuente de energía.

En resumen, la función del puerto USB consiste en proporcionar al sistema la alimentación de 5V, las señales de datos USBDM y USBDP, así como la señal de tierra provenientes del ordenador externo. Estas señales son fundamentales para su correcto funcionamiento.

Con el propósito de monitorear el funcionamiento, se han incorporado dos diodos LED (D4 y D5) que indican visualmente la actividad de la comunicación UART con el exterior. Específicamente, para los valores predeterminados de este componente, el LED D4, de color rojo, se enciende cada vez que la UART recibe datos a través del puerto USB. Por otro lado, el LED D5, de color azul, se activa cuando se envía información al PC externo. [Future Technology Devices International Ltd., 2005]

El componente principal en esta sección es la **UART** (Universal Asynchronous Receiver-Transmitter), que se utiliza para establecer comunicación en serie asincrónica con otro dispositivo. Para lograr esto, se requiere enviar un bit de inicio con nivel lógico 0 antes de transmitir los datos, indicando al receptor

que se enviará información. Los datos se transmiten desde el bit de menor peso (LSB) hasta el de mayor peso (MSB), con una longitud variable de 5 a 8 bits. Después de enviar los datos, opcionalmente se puede transmitir un bit de paridad para que el receptor pueda verificar la integridad de la comunicación. Por último, se envía un bit de parada (aunque en ocasiones pueden ser dos) con nivel lógico 1, indicando el final de la comunicación. [Gadre, 2018] [Pinacho Gómez, 2022]

En este caso, utilizaremos específicamente la **UART FT232R** de *Future Technology Devices International Ltd.* Este dispositivo tiene la capacidad de convertir la información proveniente del exterior a través del puerto USB y comunicarse con el ESP8266 mediante la UART. [Future Technology Devices International
Centrándonos en la parte de la UART, la comunicación con el ESP8266 se realizará a través de las siguientes cuatro señales:

- **TxD**: corresponde con el *buffer* dónde esperan los bits a ser transmitidos.
- **RxD**: corresponde con el *buffer* dónde esperan los bits recibidos a ser leídos.
- **RTS**: señal utilizada para *reset*.
- **DTR**: señal utilizada para poner en modo programación (*bootloader*).

Por último, se han incluido dos *jumpers* para la gestión de la alimentación. Uno de ellos se conecta directamente a la fuente de alimentación de 5V proveniente del puerto USB, mientras que el otro pasa a través de dos diodos (D2 y D3) que reducen la tensión a 3,3V. Esto se ha realizado con el fin de tener ambas tensiones disponibles en una misma placa, ya que hay componentes que requieren diferentes niveles de tensión de operación. Además, se ha incorporado el diodo D1 para impedir el flujo de corriente en sentido inverso.

4.1.2. Núcleo central: ESP8266

El módulo central del sistema es el **ESP12-F**, fabricado por el equipo **AI-Thinker**. Este módulo WiFi (figura 4.5) está basado en el procesador **ESP8266**, que cuenta con una arquitectura de 32 bits y un consumo energético reducido. Puede operar a velocidades de reloj entre 80MHz y 160MHz. El ESP8266 incluye una antena integrada, es compatible con sistemas operativos en tiempo real (RTOS) y dispone de conectividad WiFi, así como de un amplificador de baja interferencia (LNA). Cumple con los estándares IEEE802.11 b/g/n y tiene la pila de protocolo TCP/IP completamente integrada. [Ai-Thinker,] [Pascua Maestro et al., 2019]

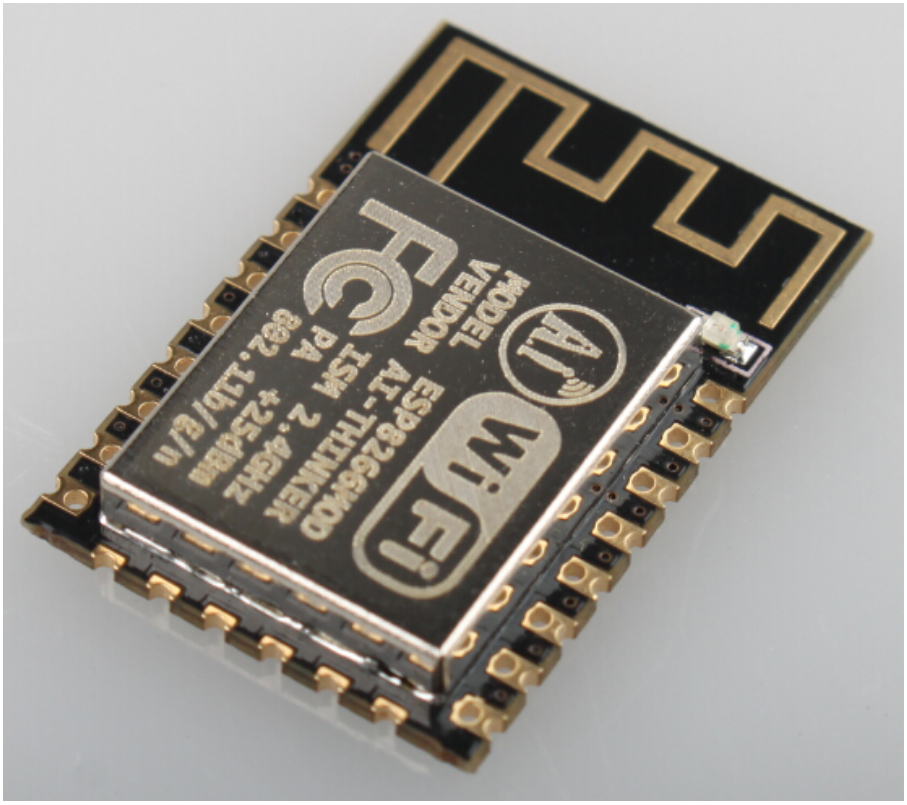


Figura 4.5: Diseño del módulo ESP12-F [Ai-Thinker,]

El módulo ESP12-F dispone de dos tipos de memoria: una memoria RAM de 36 kB y una **memoria flash externa de 4 MB**. Puede comunicarse con otros dispositivos a través de interfaces como SPI, I2C y UART. En nuestro caso, utilizaremos las interfaces SPI y UART. Esta última se utilizará para la comunicación con el PC mediante el convertidor FT232R, como se mencionó anteriormente. El protocolo SPI se empleará para la comunicación con el transceptor CAN MCP2515.[Ai-Thinker,] [Pascua Maestro et al., 2019]

Es importante destacar que el ESP8266 se programa utilizando el entorno de desarrollo **Arduino**, el cual ofrece una amplia variedad de bibliotecas que facilitan su implementación. Para el correcto funcionamiento de estas bibliotecas, se ha tenido que reconfigurar los pines asignados por defecto a la comunicación SPI, utilizando pines de propósito general (GPIO) para disponer de las comunicaciones UART y SPI de manera independiente y simultánea.[Ai-Thinker,] [Pascua Maestro et al., 2019]

Arduino

El ESP8266 se programa utilizando el entorno de desarrollo (IDE) de Arduino, que es un conjunto de herramientas diseñadas para facilitar la programación y el desarrollo de proyectos utilizando hardware Arduino [Artero, 2013].

El entorno de desarrollo de Arduino consta de tres componentes principales. En primer lugar, está el IDE de Arduino, que es una aplicación de software con una interfaz gráfica que permite a los usuarios escribir y compilar su código, así como cargarlo en la placa. En segundo lugar, el lenguaje de programación utilizado en el entorno de Arduino está basado en C/C++. Este lenguaje simplificado proporciona una forma accesible para que los usuarios programen la lógica de sus proyectos, sin tener que preocuparse demasiado por los detalles de bajo nivel. [Blum, 2019]

Por último, el entorno de desarrollo de Arduino incluye una amplia colección de bibliotecas y ejemplos. Estas bibliotecas son conjuntos de código predefinido que simplifican tareas comunes en el desarrollo de proyectos electrónicos, como la comunicación serial, el control de motores o el manejo de

sensores. Los ejemplos de código proporcionados permiten a los usuarios comprender mejor el funcionamiento de las bibliotecas y utilizarlos como punto de partida para sus propios proyectos.[Blum, 2019]

SPI: Serial Peripheral Interface

El protocolo SPI (*Serial-Peripheral Interface*) es un protocolo de comunicación ampliamente utilizado a nivel mundial y fue inventado por la compañía estadounidense Motorola. Su principal aplicación es para la comunicación entre dispositivos electrónicos. El protocolo SPI se compone de un dispositivo central llamado Maestro y uno o varios dispositivos periféricos denominados Esclavos [Saha et al., 2014]. La comunicación en el protocolo SPI se lleva a cabo mediante el uso de cuatro señales clave [Prasad and Babu, 2014] [Hernández Mangas, 2019]:

- SCK (Clock): esta señal de reloj se utiliza para sincronizar la comunicación entre el maestro y los esclavos.
- MOSI (Master Out Slave In): este bus de datos transmite los datos desde el maestro hasta el esclavo.
- MISO (Master In Slave Out): este bus de datos permite la transmisión de datos desde el esclavo hasta el maestro.
- Slave Select (SS): este bit se utiliza para seleccionar el esclavo con el cual se desea establecer la comunicación. Es un nivel lógico activo bajo.

4.1.3. Componentes CAN

En el sistema, la parte final está dedicada a la comunicación CAN. Las señales de control SPI (Serial Peripheral Interface) provenientes del ESP12-F se conectan al controlador MCP2515 (ver figura 4.6). El MCP2515 es un microcontrolador que se encarga de gestionar todas las operaciones relacionadas con la transmisión, recepción y control de las tramas enviadas y recibidas a través del bus CAN [Hernández Mangas, 2019].

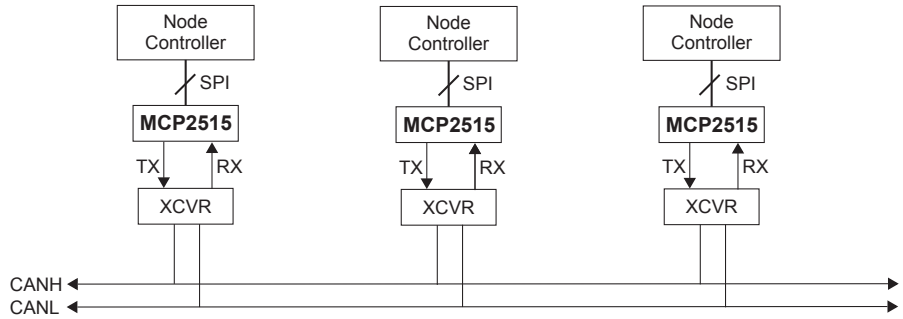
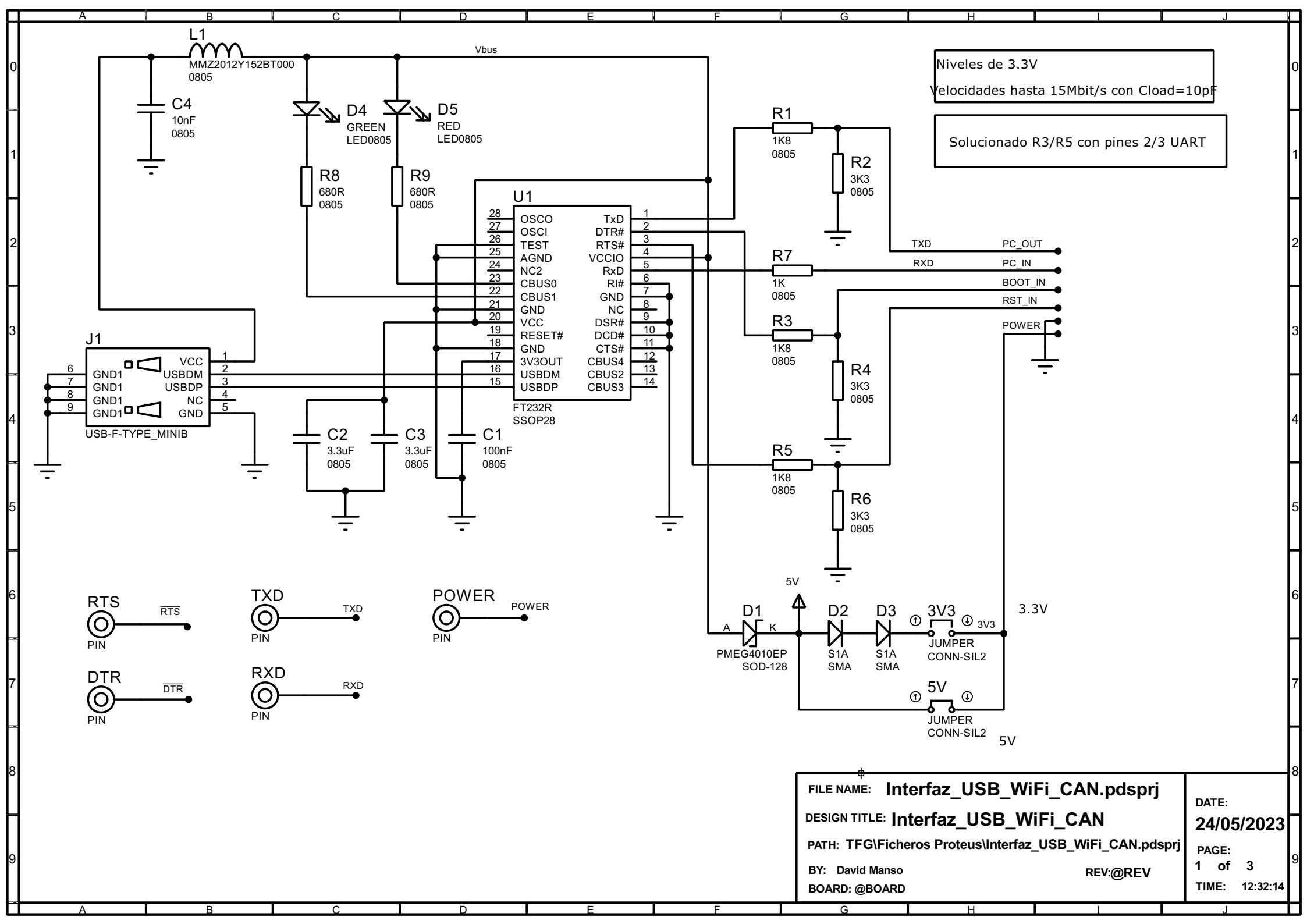


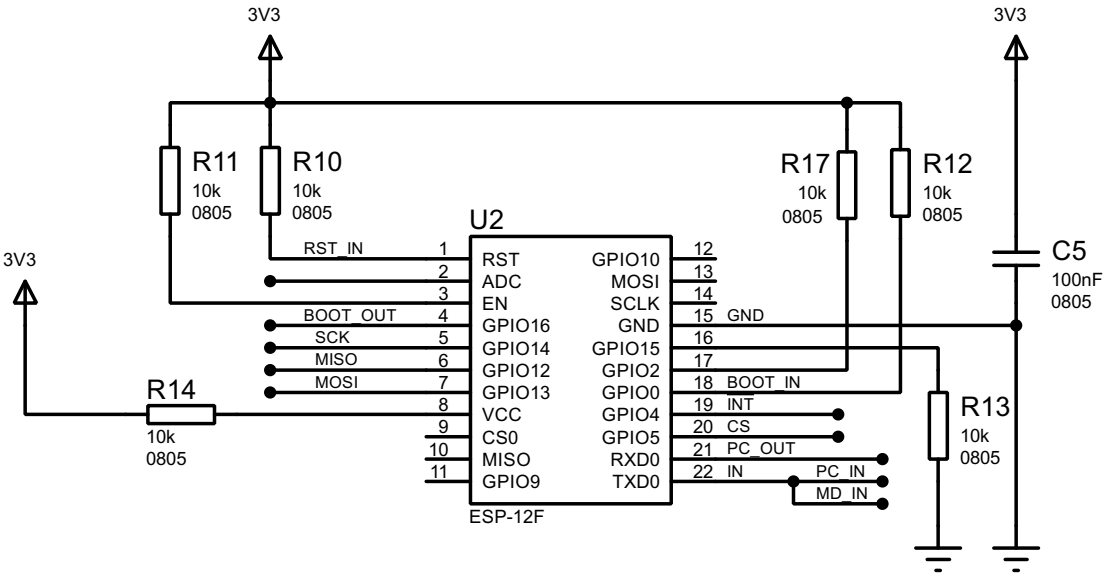
Figura 4.6: Esquema de la comunicación entre varios nodos [Hernández Mangas, 2019]

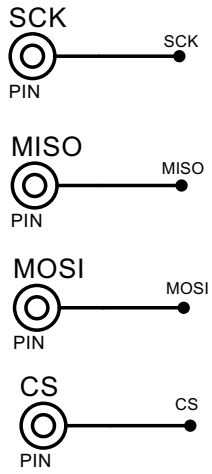
El controlador MCP2515 se comunica con el transceptor CAN MCP2551. El transceptor tiene la función de convertir una trama lógica de datos en una trama CAN. Recibe la trama de datos que el controlador desea transmitir a través del pin TXD y la envía utilizando las señales CANH y CANL. Asimismo, el transceptor realiza la función inversa, es decir, convierte las tramas recibidas en una secuencia de unos y ceros que son captadas por el controlador [Salunkhe et al., 2016].

Adicionalmente, se incorporan otros componentes en el sistema. Se utiliza un cristal de cuarzo (X1) de 20MHz para la sincronización del MCP2515. También se agrega un jumper (J4) que permite añadir una resistencia de terminación de 120Ω al bus CAN, lo cual es necesario para una correcta terminación de la línea. Por último, se cuenta con el conector J3, que actúa como la puerta de entrada/salida al bus CAN.



FILE NAME:	Interfaz_USB_WiFi_CAN.pdsprj
DESIGN TITLE:	Interfaz_USB_WiFi_CAN
PATH:	TFG\Ficheros Proteus\Interfaz_USB_WiFi_CAN.pdsprj
BY:	David Manso
BOARD:	@BOARD
REV:	@REV
DATE:	24/05/2023
PAGE:	1 of 3
TIME:	12:32:14





- Pin 21 de U2 cambiado a PC_OUT para puentear la puerta AND del esquema anterior
- Añadida R21 de 10k entre prin 17 de U2 y 3V3
- Eliminadas conexiones UART_IN, UART_OUT y SPI al ser conexiones de una misma placa
- Pin19 de U2 se conecta con interrupciones del transceptor CAN MCP2515

FILE NAME: **Interfaz_USB_WiFi_CAN.pdsprj**

DESIGN TITLE: **Interfaz_USB_WiFi_CAN**

PATH: **TFG\Ficheros Proteus\Interfaz_USB_WiFi_CAN.pdsprj**

BY: **David Manso**

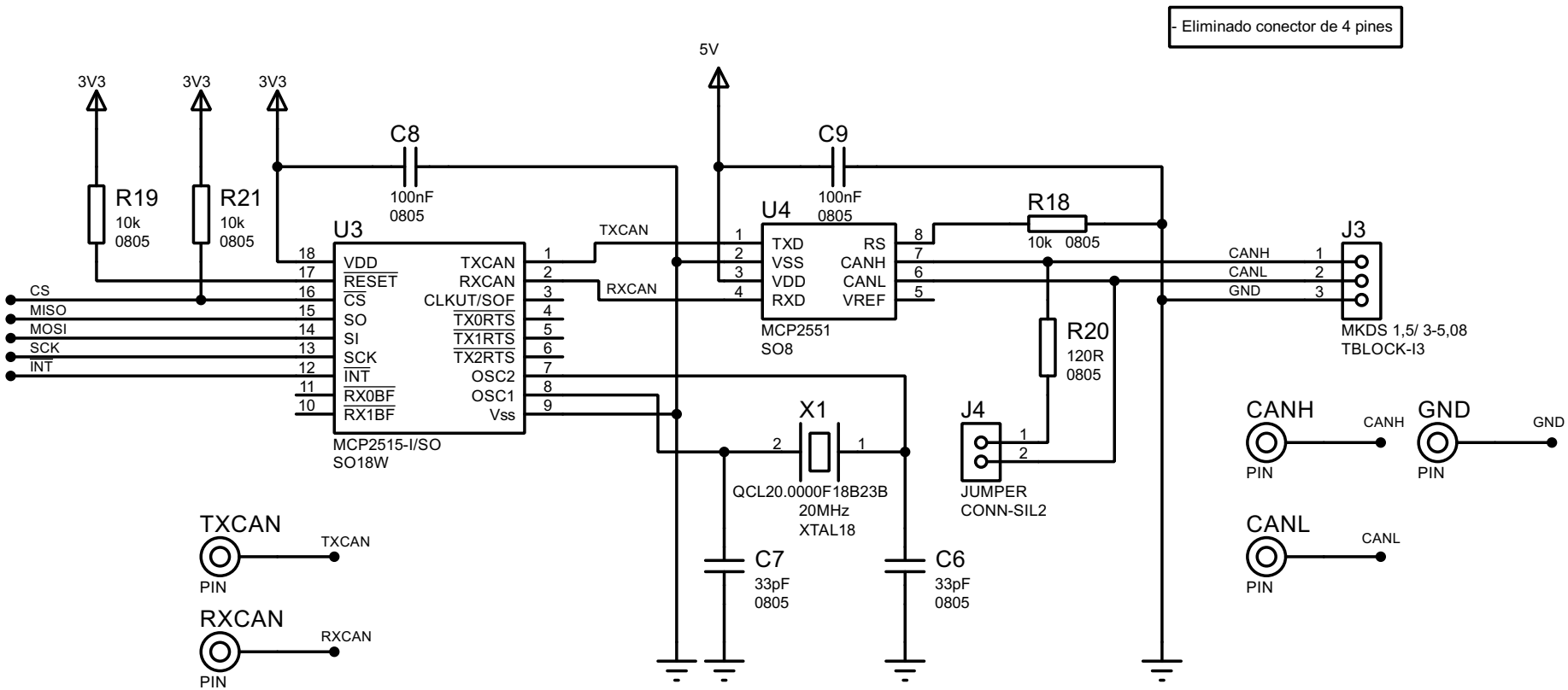
REV:**@REV**

BOARD: **@BOARD**

DATE: **24/05/2023**

PAGE: **2 of 3**

TIME: **12:32:14**



FILE NAME: Interfaz_USB_WiFi_CAN.pdsprj	DATE: 24/05/2023
DESIGN TITLE: Interfaz_USB_WiFi_CAN	PAGE: 3 of 3
PATH: TFG\Ficheros Proteus\Interfaz_USB_WiFi_CAN.pdsprj	
BY: David Manso	REV:@REV
BOARD: @BOARD	TIME: 12:32:14

4.2. Software

El objetivo principal es desarrollar un software que cumpla con los requisitos del proyecto y sea compatible tanto con LPC2103 como con ESP12-F. La mayoría de las funciones y rutinas son comunes para ambos dispositivos, pero existen diferencias en el código debido a las características internas de los módulos, como las direcciones de memoria y los métodos de control de las comunicaciones SPI y UART. Cabe destacar que cada dispositivo tiene un uso diferente: el sistema que incluye el chip LPC2103 actúa como cliente y el del módulo ESP12-F como servidor. Aunque se consideró la posibilidad de integrar ambas funciones en cada dispositivo, finalmente se optó por no hacerlo debido a limitaciones de capacidad y eficiencia.

A continuación, se presentan las funciones comunes para ambos sistemas, específicamente las funciones relacionadas con el controlador CAN MCP2515, ya que este controlador es el mismo en ambos sistemas. Estas funciones, cuyo código se encuentra en los anexos, son las siguientes:

- **MCP2515_Reset()**: envía los comandos necesarios a través de SPI para reiniciar el controlador.
- **MCP2515_Read(addr)**: devuelve el contenido del registro ubicado en la dirección *addr*.
- **MCP2515_Write(addr, data)**: escribe el valor *data* en el registro de memoria en la dirección *addr*.
- **MCP2515_ReadStatus()**: devuelve el contenido del registro STATUS.
- **MCP2515_RTS(buffer)**: envía el *buffer* especificado.
- **CAN_Config(speed)**: configura el controlador con la velocidad *speed* indicada y establece los demás registros en sus valores predeterminados.
- **CAN_Filter(mk, id, ide)**: escribe en los registros destinados al filtrado de tramas el contenido necesario a partir de la máscara *mk*, el identificador *id* y si es una trama extendida *ide*.
- **CAN_Transmit(CAN_Message *Msg)**: función encargada de transmitir el mensaje CAN *msg*. Escribe en los registros de transmisión y llama a la función MCP2515_RTS() para enviar la trama.

Además, se destaca la función *txTrama()*, que se encarga de transmitir una trama. A continuación se muestra su código en lenguaje C:

```
void txTrama(char op, const char msg[], int trama, int len)
{
    int t_init, ID_final;
    CAN_Message Mensaje;

    ID_final = (id_propio << 9) | (op << 5) | trama;
    Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : len, data : {msg[0 +
        (8 * trama)], msg[1 + (8 * trama)], msg[2 + (8 * trama)], msg[3 + (8 *
        trama)], msg[4 + (8 * trama)], msg[5 + (8 * trama)], msg[6 + (8 * trama)],
         msg[7 + (8 * trama)]}};

    CAN_Transmit(&Mensaje);
    if (STATE == 1)
        return;

    t_init = millis();
    while (tramaRecibida == 0)
    {
        wdt_reset();
        if (millis() > (t_init + 1000))
        {
            STATE = 1;
            break;
```

```
        }
    }
    tramaRecibida = 0;
}
```

En esta función, se forma el identificador final siguiendo las pautas descritas en la sección 3.3, utilizando los parámetros proporcionados por la función que llama a esta. La variable *id_propio* es una variable global que se define al inicio del programa. Luego se crea la estructura del mensaje CAN, donde se incluye el identificador final previamente formado, se indica si es una trama remota y se agrega tanto la longitud de los datos de la trama como los propios datos que se desean transmitir (*msg*) con el desplazamiento correspondiente al número de trama. Una vez creado el mensaje, se envía utilizando la función CAN_Transmit(). Por último, se verifica si la transmisión fue exitosa y, en caso contrario se iguala *STATE* a 1 y se sale de la función. A continuación, se espera a recibir una trama de asentimiento por parte del receptor. Si no se recibe ninguna trama durante un segundo, también sale de la función con STATE=1. Es importante mencionar que la forma de medir los tiempos son diferentes para Arduino y LPC en función de las librerías empleadas.

4.3. Arduino

En esta sección se hablará del código Arduino del ESP12-F. En concreto nos centraremos en los pasos principales para después adentrarnos en las funciones particulares para el sistema de ficheros y la conexión HTTPS. Cualquier código Arduino consta de dos partes principales: la función *setup()* y la función *loop()*. La función *setup()* se ejecuta una vez al comienzo del programa y se encarga de la configuración inicial. A continuación se muestra el código correspondiente:

```
void setup()
{
    Serial.begin(baud);
    SPI_Init();             // Inicializo el SPI
    CAN_Config(CAN_Speed); // Configuro MCP2515

    _printf("MCP configurado correctamente. \n");

    bool result = SPIFFS.begin();  // Se monta el sistema de ficheros
    if (result)
        _printf("SPIFFS montado correctamente\n");

    WiFi.mode(WIFI_STA);  // Realizo la conexión a la red WiFi

    WiFiMulti.addAP("APN", "PWD");  // Se agregan los nombres de puntos de acceso
        y contraseñas

    Serial.println("Conectando...\n");

    if (WiFiMulti.run() == WL_CONNECTED)
        Serial.println("Conectado\n");

    attachInterrupt(4, isr, FALLING);  // Activo las interrupciones en el pin
        GPIO4
}
```

En la función *setup()* descrita anteriormente, se inicia la comunicación con el puerto serie con el objetivo de monitorizar el sistema. A continuación, se configura e inicia la comunicación SPI además de la configuración inicial del microcontrolador MCP2515 y se muestra un mensaje indicando el éxito de esta. Entonces, se inicia el sistema de ficheros SPIFFS del ESP8266 y si la inicialización es exitosa, se muestra un mensaje correspondiente.

Después, se configura el modo de conexión WiFi y se agregan los nombres de los puntos de acceso y las contraseñas correspondientes. Por último, se verifica si la conexión WiFi se establece correctamente utilizando *WiFiMulti.run()* y, si es así, se muestra un mensaje indicando que la conexión se ha

realizado con éxito. Además, se activan las interrupciones en el pin GPIO4 mediante *attachInterrupt()* para recibir las notificaciones de llegada de tramas al MCP2515.

La función *loop()* se ejecuta de forma continua. En cada inicio, se establecen las variables globales en sus valores predeterminados. Luego, se vuelve a configurar el MCP2515 para evitar problemas de ejecuciones anteriores. Una vez configurado, el programa permanece en un bucle escuchando constantemente hasta que se reciba una trama con el código de operación *OP_Request*. Si no se recibe nada durante 5 segundos, se reinicia el MCP2515 para evitar problemas ocultos al usuario. Cuando se recibe una trama, se activa el filtrado del controlador para recibir solo tramas con el identificador de la solicitud, evitando las interferencias del resto de dispositivos del bus.

```
void loop()
{
inicio:
    OPCODE = 1;
    STATE = 0;
    filtro = 0; // Se deshabilita el filtro. Permitimos mensajes de cualquier
        cliente
    _printf(".\n");
    MCP2515_Write(MCP_CANINTF, 0x00); // Borro notificaciones

    CAN_Config(CAN_Speed);
    _printf("\n\nEscuchando...\n");

    int t_actual = millis(); // Espero a que me llegue una solicitud y deshabilito
         el WDT
    while (tramaRecibida == 0 && OPCODE != OP_Request)
    {
        wdt_reset();
        if (millis() > (t_actual + 5000)) // Si no recibo nada en 5 segundos,
            reseteo los errores
        {
            CAN_Config(CAN_Speed);
            t_actual = millis();
        }
    }

    _printf("Recibo solicitud de %d\n", id_ajeno);

    filtro = 1; // Filtro la recepción para recibir solo tramas con el mismo
        identificador
    id_filtro = id_ajeno;
    CAN_Config(CAN_Speed);

Other_Request:
    tramaRecibida = 0;
    sprintf(id_a, "%d", id_ajeno);
    enviarMsg(OP_RequestOk, id_a, 0);

    GestorRecepcion();

    if (strncmp("http", txt, 4) == 0) // Comprobar si es una URL
    {
        NavegarWeb(txt);
        goto inicio;
    }
    else // Es una ruta de archivo en el sistema de archivos
    {
        LeerFile(txt);
        goto inicio;
    }
}
```

Después, el programa espera recibir la solicitud del cliente y, según el tipo de solicitud, que puede ser un archivo en el sistema de archivos previamente inicializado o una URL que requiere acceso a Internet. La recepción de la solicitud se realiza en la función *GestorRecepcion()* y se almacena en la variable **txt**. Luego, se compara esta variable con la cadena "*http*"para determinar si se trata de una URL o una ruta en el sistema de archivos. En función del resultado de la comparación se llama a una de las siguientes funciones:

- ***NavegarWeb(txt)***: esta función realiza todos los pasos necesarios para la descarga del código HTML de la pagina Web solicitada por el cliente.
- ***LeerFile(txt)***: esta función realiza todos los pasos necesarios para la lectura del fichero solicitado por el cliente dentro del sistema de ficheros.

4.3.1. Sistema de Ficheros en ESP8266

Para establecer un sistema de archivos en un ESP8266, es necesario crear una carpeta llamada *data* en el mismo directorio que el programa principal (.ino). En esta carpeta, se pueden incluir los archivos que se deseen que estén disponibles en el servidor. Una vez creada la carpeta, se deben cargar estos archivos en el sistema de archivos del ESP12-F desde el IDE de Arduino. A partir de ese momento, los ficheros estarán listos para ser usados.

Supongamos que se recibe una solicitud con una ruta de un fichero. En ese caso, el servidor llama a la función *LeerFile()*, que se muestra a continuación:

```
void LeerFile(txt) {
    // Lectura del archivo
    strcpy(path, "/");
    strcat(path, txt);

    File file = SPIFFS.open(path, "r");
    if (!file) {
        Serial.println("File not found");
        enviarMsg(OP_NoFile, "", 0);
        file.close();
        return;
    }

    int archivoSize = file.size();
    int blockSize = 1024;

    int numBloques = (archivoSize + blockSize - 1) / blockSize;

    for (int i = 0; i < numBloques; i++) {
        char buffer[blockSize] = {0};
        int bytesLeidos = file.readBytes(buffer, blockSize);

        if (i == (numBloques - 1)) {
            enviarWeb(OP_FileEnd, buffer);
        } else {
            enviarWeb(OP_File, buffer);
            if (STATE == 1) {
                _printf("Fallo de transmision\n");
                break;
            }
        }
    }

    file.close();
    _printf("Se cierra el fichero\n");

    goto inicio;
}
```

En esta función, se agrega el carácter '/' al principio de la ruta recibida para acceder al sistema de ficheros. Luego, se intenta abrir el archivo en modo de lectura ('r'). Si el archivo no se encuentra, se envía una trama al cliente indicando que no se encontró el archivo (*OP_NoFile*), se cierra el archivo y se vuelve al estado inicial del servidor.

Después, se obtiene el tamaño completo del archivo y se calcula el número de bloques de 1024 bytes necesarios para cubrir todo el archivo. A continuación, se procede a leer el archivo de bloque en bloque mientras se transmiten. Esta acción se realiza de manera continuada hasta llegar al último bloque, momento en el que la última trama del bloque leído se envía con el opcode *OP_FileEnd* para indicar el final del archivo. Se verifica que la transmisión se realice correctamente y, en caso de haber algún error, se aborta la transmisión. Finalmente, se cierra el fichero y se vuelve al estado inicial del servidor.

4.3.2. Conexión HTTPS con ESP8266

La función *NavegarWeb()* se utiliza cuando se recibe una URL de una página web en lugar de una ruta del sistema de ficheros. Esta función establece una conexión segura utilizando SSL/TLS, a pesar de que se desactiva la verificación de los certificados SSL. A continuación, se inicia la conexión a la URL indicada y se muestra por pantalla el código de respuesta de la página.

```
void NavegarWebPrueba(String url)
{
    // proporciona una conexión segura utilizando SSL/TLS.
    std::unique_ptr<BearSSL::WiFiClientSecure> client(new BearSSL::
        WiFiClientSecure);

    bool mfln = client->probeMaxFragmentLength("tls.mbed.org", 443, 1024);
    // Serial.printf("\nConnecting to https://www.google.es/\n");
    // Serial.printf("Maximum fragment Length negotiation supported: %s\n", mfln ?
        "yes" : "no");
    // if (mfln)
    // {
    //     client->setBufferSizes(1024, 1024);
    // }

    Serial.print("[HTTPS] begin...\n");

    // Se desactiva la verificación de certificados SSL
    client->setInsecure();

    HTTPClient https;

    if (https.begin(*client, url))
    {

        Serial.print("[HTTPS] GET...\n");
        // start connection and send HTTP header
        int httpCode = https.GET();
        if (httpCode > 0)
        {
            // HTTP header has been send and Server response header has been
                handled
            Serial.printf("[HTTPS] GET... code: %d\n", httpCode);

            // file found at server
            if (httpCode == HTTP_CODE_OK)
            {

                // get length of document (is -1 when Server sends no Content-
                    Length header)
                int len = https.getSize();
```

```
                // read all data from server
                while (https.connected() && (len > 0 || len == -1))
                {
                    // create buffer for read
                    char buff[1024] = {0};

                    // get available data size
                    size_t size = client->available();

                    if (size)
                    {
                        // read up to buff size
                        int c = client->readBytes(buff, ((size > sizeof(buff)) ?
                            sizeof(buff) : size));

                        char *last5 = strrchr(buff, '\0') - 7;

                        if (strstr(buff, "</html>") != NULL)
                        {
                            enviarWeb(OP_FileEnd, buff);
                            goto FINNISH_WEB;
                        }
                        else
                            enviarWeb(OP_File, buff);
                    }
                }
                STATE = 3;
                txTrama(OP_NoInternet, "", 0, 0);
            }
        }
        else
        {
            Serial.printf("[HTTPS] GET... failed, error: %s\n", https.
                errorToString(httpCode).c_str());
            STATE = 3;
            txTrama(OP_NoInternet, "", 0, 0);
        }
FINNISH_WEB:
        https.end();
    }
    else
    {
        Serial.printf("Unable to connect\n");
        STATE = 3;
        txTrama(OP_NoInternet, "", 0, 0);
    }
}
```

Una vez se ha conseguido acceder a la página web, se lee la información en bloques de 1024 y de estos, se comparan los últimos 7 caracteres con la cadena "</**html**>" para averiguar si se trata del final del código HTML y poder enviar el opcode *OP_FileEnd* en la última trama del bloque leído. En los casos que no se pueda conectar a la pagina web se envía una trama de información al cliente y regresa al estado inicial, habiendo intentado la conexión al punto de acceso previamente para evitar posibles fallos de conexión.

4.4. LPC2103

Centrándonos en la parte del cliente, se utiliza un sistema completo con un microcontrolador principal basado en el chip ARM LPC2103. Al inicio de la ejecución, se configura el microprocesador para establecer las entradas y salidas requeridas para la comunicación SPI con el resto de dispositivos del sistema, así como para manejar interrupciones. Además, al igual que con el ESP12-F, se realiza la inicialización del controlador CAN, la comunicación SPI y la UART del sistema para permitir la

visualización de datos en el terminal. Esta última función es similar a la función *Serial.baud()* en la programación de Arduino. Por último, se habilitan las interrupciones y se muestra el título del programa en la pantalla.

```
        PINSEL0 = 0x00001505;    // RXD0,TXD0 con la UART
                                 // SCK0, MISO0, MOSI0 seleccionadas, SSEL0 no
        PINSEL1 = 0x00000001;    // P0.16 EINT0 activo
        IODIR |= (1 << 21) | (1 << 20); // P0.21,P0.22 are outputs
        IODIR |= CAN_CS | USB_SS | SPI_CS;       // P0.7 is output
        IOPIN = 0xFFFFFFFF;              // All outputs highxx
        IOCLR = (1 << 20);

        UART0_Init();
        SPI_Init();
        CAN_Config(CAN_SPEED); // Configuro MCP2515

        EXTMODE = 1;  // EINT0 edge sentitive
        EXTPOLAR = 0; // EINT0 falling edge

        VICIntEnable = (1 << EINT0_INT);
        VICVectAddr2 = (unsigned int)CAN_ISR;
        VICVectCntl2 = (1 << 5) | EINT0_INT;

        // Habilito interrupciones IRQ
        asm volatile("mrs r0,cpsr\n"
                                 "bic r0,r0,#0x80\n"
                                 "msr cpsr,r0");

        UART0_print("\t\tCLIENTE INTERFAZ WIFI - CAN\n");
```

Es importante mencionar que el cliente está programado para filtrar todas las tramas que no tengan el identificador del servidor, lo que significa que solo puede recibir tramas del servidor. Esta restricción se incluye para evitar posibles variaciones en la máquina de estados del dispositivo y prevenir errores en la recepción de archivos. De esta manera, si el cliente está en medio de la recepción de los datos y detecta una trama de otro dispositivo, se descarta para garantizar la integridad en el proceso de recepción.

4.4.1. Interfaz para el usuario

Para visualizar el funcionamiento del programa, se ha diseñado una interfaz de demostración que refleja junto al título mencionado anteriormente cuatro opciones diferentes para solicitar archivos al servidor. La interfaz se representa en la figura 4.7.

```
Filtro activado con las siguientes caracteristicas:
Mascara = FFFFFE00, Identificador = 7711
MCP configurado correctamente.
                CLIENTE INTERFAZ WIFI - CAN

Selecciona una de las siguientes opciones:
        => 1: Solicita fichero Formula1.txt
        => 2: Solicita fichero MCP2515.pdf
        => 3: Solicita codigo html tel.uva.es
        => 4: Solicita los datos anteriores de manera continuada
```

Figura 4.7: Ejemplo de la interfaz en el terminal

El usuario del cliente debe seleccionar la tecla correspondiente al recurso que desea obtener. Una vez seleccionada, el sistema se encargará de todo el proceso y mostrará por pantalla el contenido del

recurso solicitado. En caso de algún error comunicado por el servidor, se mostrará la información del error. Se ha optado por mostrar el contenido directamente en pantalla en lugar de almacenarlo debido a las limitaciones de memoria de este dispositivo.

Líneas Futuras

Aunque el proyecto presenta un estado bastante completo, hay que ser consciente de que este tiene mucho margen de mejora. Por lo tanto, para el futuro, se proponen las siguientes líneas de trabajo:

- Realizar una nueva fabricación de la PCB corrigiendo los errores ya identificados en la versión actual.
- Mejorar el servidor para que pueda procesar la información de una página web, comprimir los archivos y analizar códigos JSON y XML. Esto permitirá enviar al cliente un menor número de tramas, logrando una mayor eficiencia en la transmisión de datos.
- Incrementar el tamaño de memoria disponible con el fin de poder almacenar ficheros de mayor tamaño.
- Desarrollar una aplicación o una solución similar para controlar el módulo ESP8266 y permitir que envíe información a los dispositivos del bus sin que estos lo soliciten. Esta funcionalidad resulta útil para transmitir información de emergencia o realizar actualizaciones cruciales.
- Revisar y ajustar los códigos de operación utilizados, adaptándolos a procesos más específicos o reemplazándolos según las necesidades identificadas.

Bibliografía

[Ai-Thinker,] Ai-Thinker. *ESP-12F WiFi Module. Version 1.0.*

[Artero, 2013] Artero, Ó. T. (2013). *ARDUINO. Curso práctico de formación.* RC libros.

[Blum, 2019] Blum, J. (2019). *Exploring arduino: Tools and techniques for engineering wizardry (2a ed.).* John Wiley.

[Defaz Andrango, 2007] Defaz Andrango, M. L. (2007). Estudio del protocolo can (controller area network) y su aplicación en redes de control. B.S. thesis, QUITO/EPN/2007.

[Future Technology Devices International Ltd., 2005] Future Technology Devices International Ltd. (2005). *FT232R USB UART I.C.*

[Gadre, 2018] Gadre, D. V. (2018). *Getting Started with Tiva ARM Cortex M4 Microcontrollers A Lab Manual for Tiva LaunchPad Evaluation Kit / by Dhananjay V. Gadre, Sarthak Gupta.* Springer India, New Delhi, 1st ed. 2018. edition.

[Hernández Mangas, 2019] Hernández Mangas, J. M. (2019). *Interconexión de Sistemas Digitales.* I.T.E. Telecomunicación.

[Microchip Technology Inc., 2019] Microchip Technology Inc. (2019). *MCP2515. Stand-Alone CAN Controller with SPI Interface.*

[Morales and Antonio, 2005] Morales, C. and Antonio, C. (2005). Desarrollo de un sistema educativo para la enseñanza del protocolo de comunicaciones can. *Universidad Tecnológica de la Mixteca, Huajapan de León-Abril.*

[Pascua Maestro et al., 2019] Pascua Maestro, I. et al. (2019). Sistema electrónico de apertura de puerta de garaje desde terminal smartphone.

[Pinacho Gómez, 2022] Pinacho Gómez, R. (2022). *TEMA 3: Unidades de comunicación y tratamiento de la señal.* Dpto. de Electricidad y Electrónica. Universidad de Valladolid.

[Prasad and Babu, 2014] Prasad, T. D. and Babu, B. R. (2014). Design and simulation of spi master/slave using verilog hdl. *Int. J. Sci. Res*, 3(8).

[Saha et al., 2014] Saha, S., Rahman, M. A., and Thakur, A. (2014). Design and implementation of spi bus protocol with built-in-self-test capability over fpga. In *2014 International Conference on Electrical Engineering and Information & Communication Technology*, pages 1–6. IEEE.

[Salunkhe et al., 2016] Salunkhe, A. A., Kamble, P. P., and Jadhav, R. (2016). Design and implementation of can bus protocol for monitoring vehicle parameters. In *2016 IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology (RTEICT)*, pages 301–304.

[Sánchez Vela et al., 2016] Sánchez Vela, L., Molano Clemente, M., Fabela Gallegos, M., Martínez Madrid, M., Hernández Jiménez, J., Vázquez Vega, D., and Flores Centeno, O. (2016). Revisión documental del protocolo can como herramienta de comunicación y aplicación en vehículos. *Publicación técnica*, (474).

[Sánchez Arias, 2018] Sánchez Arias, O. (2018). Protocol de comunicació can.

[Velázquez, 2015] Velázquez, C. (2015). Gauss y el mito de los primeros cien números.

Capítulo 5

Anexo I:

5.1. Código fuente para los dispositivos con el chip LPC2103

5.1.1. Función main

```
// ========================================================
// Interfaz WiFi - CAN
// Autor: David Manso Fernandez
// Tutor: Jesus M. Hernandez Mangas
// Fecha: Mayo 2023
// Trabajo de Fin de Grado
// ========================================================

#ifndef __MAIN_C
#define __MAIN_C

#include "system.h"
#include "CAN.c"
#include "funciones.c"
#include "UART.c"
#include "SPI.c"
#include "system.c"

#define EINT0_INT (14) // VIC register
void CAN_ISR(void) __attribute__((interrupt("IRQ")));

int filtro = 1;
int id_filtro = 7711;
int mk_filtro = 0xFFFFFE00; // Esta mascara acepta todas las tramas que tengan el
    id_filtro
int ide_filtro = 1;

char OPCODE;
int id_propio = 1230;
int id_ajeno;
int n_trama;

volatile int STATE = 0;
char txt[1000] = {0};
volatile int tramaRecibida;
char fifo[MAX_SIZE_FIFO];
int write_ptr = 0;
int read_ptr = 0;

void CAN_ISR(void)
{
        tramaRecibida = 1;
        escucha();
        MCP2515_Write(MCP_CANINTF, 0x00); // Borro notificaciones
        VICVectAddr = -1;
```

```
        EXTINT = 1; // Borro flag de registro de notificacion de interrupciones.
}

char *menutxt ="\n"
"Selecciona una de las siguientes opciones:     \n"
"       => 1: Solicita fichero Formula1.txt\n"
"       => 2: Solicita fichero PCB_top.pdf\n"
"       => 3: Solicita codigo html tel.uva.es\n"
"       => 4: Solicita los datos anteriores de manera continuada\n"
"\n\n";


void main()
{

        PINSEL0 = 0x00001505;   // RXD0,TXD0 con la UART
                                // SCK0, MISO0, MOSI0 seleccionadas, SSEL0 no
        PINSEL1 = 0x00000001;   // P0.16 EINT0 activo
        IODIR |= (1 << 21) | (1 << 20); // P0.21,P0.22 are outputs
        IODIR |= CAN_CS | USB_SS | SPI_CS;      // P0.7 is output
        IOPIN = 0xFFFFFFFF;             // All outputs high
        IOCLR = (1 << 20);

        UART0_Init();
        UART0_print("\t\tCLIENTE INTERFAZ WIFI - CAN\n");

        SPI_Init();
        CAN_Config(CAN_SPEED); // Configuro MCP2515

        EXTMODE = 1;  // EINT0 edge sentitive
        EXTPOLAR = 0; // EINT0 falling edge

        VICIntEnable = (1 << EINT0_INT);
        VICVectAddr2 = (unsigned int)CAN_ISR;
        VICVectCntl2 = (1 << 5) | EINT0_INT;

        // Habilito interrupciones IRQ
        asm volatile("mrs r0,cpsr\n"
                                "bic r0,r0,#0x80\n"
                                "msr cpsr,r0");


        while (1)
        {
                UART0_print(menutxt); // Se muestra el menu de opciones
                char cmd = _getch();    // Captura la tecla pulsada
                        switch (cmd){
                                case '1':
                                        solicitarFichero("Formula1.txt");
                                break;
                                case '2':
                                        // solicitarFichero("MCP2515.pdf");
                                        solicitarFichero("PCB_top.PDF");
                                break;
                                case '3':
                                        solicitarFichero("https://www.tel.uva.es/"
                                            );
                                break;
                                case '4':
                                        solicitarFichero("Formula1.txt");
                                        _delay_ms(2000);
                                        solicitarFichero("PCB_top.PDF");
                                        _delay_ms(2000);
                                        solicitarFichero("https://www.tel.uva.es/"
                                            );
                                break;
```

```
                        }


        }
}

#endif
```

5.1.2. Resto de funciones complementarias

```
#include "system.h"
/*****************************************************/
/*              Rutinas de interrupcion              */
/*****************************************************/

volatile ms, sec;
int aux = 0;

void IRQ_Timer0_Routine(void) __attribute__((interrupt("IRQ")));
void IRQ_Spurious_Routine(void) __attribute__((interrupt("IRQ")));

void IRQ_Timer0_Routine(void)
{
    aux++;
    ms++;
    if (ms == 1000)
    {
        aux = 0;
        ms = 0;
        sec++;
    };
    T0IR = 1;          // Borramos interrupcion en TIMER0
    VICVectAddr = -1; // EOI para el controlador de interrupciones
}
void IRQ_Spurious_Routine(void)
{
    _printf("\r\n--- IRQ, Spurious interrupt\r\n");
    VICVectAddr = -1; // End of interrupt: update priority logic
}

void TIMER0_Init()
{
    // T0TC = 0;                      // Timer counter 0
    T0PC = 0;                       // Prescaler counter 0
    T0PR = (1e-3 * PCLK - 1) / 2; // Timer value aprox 1ms
    T0TCR = 0b00000011;             //
    //                .          Enable = 1
    //               .           Reset = 1
    //        ......             Reserved
    T0CTCR = 0b00000000;
    //               ..              Timer mode (00)
    //              ..               Clock source in counter mode non-sense
    //          ....               Reserved
    T0MR0 = 1; // Timer value aprox (T0MR0 + 1)*TT. 2veces TT[index]/2
    T0MCR = 0b0000000000000011;
    //                      ... MR0 control 011 non-stop, reset, interrupt
    //                   ...    MR1 control 000 non-stop, non reset, non interrupt
    //                ...       MR2 control 000 non-stop, non reset, non interrupt
    //             ...          MR3 control 000 non-stop, non reset, non interrupt

    // Interrupt setup
    VICIntEnable = (1 << 4);                                // Timer 0 interrupt
        enable
    VICDefVectAddr = (unsigned int)IRQ_Spurious_Routine; // unvectored IRQs
```

```
    VICVectAddr0 = (unsigned int)IRQ_Timer0_Routine;      // Vector for slot 0
    VICVectCntl0 = (1 << 5) | 4;                          // Slot 0 for Timer0 and
        enabled

    // Enable both IRQ interrupts
    asm volatile("mrs r0,cpsr\n"
                 "bic r0,r0,#0x80\n"
                 "msr cpsr,r0");
    T0TCR = 0b00000001; //
                        //                .          Enable = 1
                        //               .           Reset = 0
                        //        ......             Reserved
}

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


void write_FIFO(char *data)
{
    char *p = data;
    while (*p != '\0')
    {
        int next_write_ptr = (write_ptr + 1) % MAX_SIZE_FIFO;
        if (next_write_ptr != read_ptr)
        {
            fifo[write_ptr] = *p;
            write_ptr = next_write_ptr;
        }
        p++;
    }
}

char *read_FIFO()
{
    static char data[MAX_SIZE_FIFO + 1];
    char *p = data;
    while (read_ptr != write_ptr)
    {
        *p = fifo[read_ptr];
        read_ptr = (read_ptr + 1) % MAX_SIZE_FIFO;
        if (*p == '\0')
        {
            break;
        }
        p++;
    }
    *p = '\0';
    return data;
}

/**
 * @brief Funcion que se encarga de enviar una trama con la lognitud deseada. Util
     para enviar ACK, Request etc.
 *
 * @param op OPCODE a transmitir
 * @param msg msg a transmitir si se quiere
 * @param len longitud de la trama a transmitir
 */
void enviarTrama(char op, const char msg[], int len)
{
    int ID_final;
    CAN_Message Mensaje;

    ID_final = (id_propio << 9) | (op << 5) | 0;
```

```
    Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : len, data : {msg[0],
        msg[1], msg[2], msg[3], msg[4], msg[5], msg[6], msg[7]}};
    CAN_Transmit(&Mensaje);
    if (STATE == 1)
        _printf("to request\n");
}
void enviarTramaRequest(char op, const char msg[], int len)
{
    int ID_final;
    CAN_Message Mensaje;

    ID_final = (id_propio << 9) | (op << 5) | 0;
    Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : len, data : {msg[0],
        msg[1], msg[2], msg[3], msg[4], msg[5], msg[6], msg[7]}};
    CAN_TransmitRequest(&Mensaje);

    if (STATE == 1)
        _printf("to request\n");
}

/**
 * @brief Se encarga de decodificar la trama recibida asignando el OPCODE y el ID
     del transmior.
 */
void escucha()
{
    char SIDH, SIDL, EID8, EID0, DLC, status, remote, type;
    int i;

    if (((status >> 4) & 0x01) == 1)
        type = 1; // Extended identifier
    else
        type = 0; // Standard identifier
    if (((status >> 3) & 0x01) == 1)
        remote = 1; // Standar identifier Remote frame
    else
        remote = 0; // Extended identifier Remote Data frame

    i = MCP2515_ReadStatus();
    if (i & 0x01) // Buffer 0
    {

        SIDH = MCP2515_Read(MCP_RXB0SIDH);      // Standard identifier high
        SIDL = MCP2515_Read(MCP_RXB0SIDL);      // Standard identifier low
        EID8 = MCP2515_Read(MCP_RXB0EID8);      // Extended identifier high
        EID0 = MCP2515_Read(MCP_RXB0EID0);      // Extended identifier low
        DLC = MCP2515_Read(MCP_RXB0DLC) & 0xF; // Data length code

        n_trama = EID0 & 0b11111;

        for (i = 0; i < DLC; i++)
            txt[i + (8 * n_trama)] = MCP2515_Read(MCP_RXB0D0 + i);
    }
    else // Buffer 1
    {
        SIDH = MCP2515_Read(MCP_RXB1SIDH);      // Standard identifier high
        SIDL = MCP2515_Read(MCP_RXB1SIDL);      // Standard identifier low
        EID8 = MCP2515_Read(MCP_RXB1EID8);      // Extended identifier high
        EID0 = MCP2515_Read(MCP_RXB1EID0);      // Extended identifier low
        DLC = MCP2515_Read(MCP_RXB1DLC) & 0xF; // Data length code

        n_trama = EID0 & 0b11111;

        for (i = 0; i < DLC; i++)
            txt[i + (8 * n_trama)] = MCP2515_Read(MCP_RXB1D0 + i);
    }
```

```
    OPCODE = ((EID8 & 0b1) << 3) | (EID0 >> 5);
    id_ajeno = (EID8 >> 1) | ((SIDL & 0b11) << 7) | ((SIDL >> 5) << 9) | (SIDH <<
        12);
}

/**
 * @brief Esta funcion es la encargada de recibir las tramas y comprobar si son
     correctas una vez se haya establecido la conexion con el otro terminal. Una
     vez comprobado que la trama es correcta regresamos a la funcion principal.
 */
void GestorRecepcion()
{
    int checkFrame = 0;
    int n_trama_max = 0;
    int paquete = 0;
    int i = 0;
reenvio:
    _memset(txt, '\0', max_MSGlength);
    ms = 0;
    sec = 0;
    checkFrame = 0;
    n_trama_max = 0;
    // tramaRecibida = 0;
    STATE = 0;

    while (1)
    {
    listen:
        TIMER0_Init();
        // Esperamos a recibir una trama.
        // Si tarda mas de 15 segundos se aborta comunicacion.
        // Se pone tanto tiempo porque el servidor requiere de tiempo para
            conectarse con Internet
        while (tramaRecibida == 0)
        {
            if (sec > 15)
            {
                ms = 0;
                sec = 0;
                STATE = 1;
                _printf("Timeout RX\n");
                goto fin_timeout;
            }
        }

        checkFrame += n_trama; // Sumamos el n_trama de las tramas recibidas para
            hacer checksum con las tramas.
        if (n_trama > n_trama_max)
            n_trama_max = n_trama;

        if (OPCODE == OP_FilePartEnd || OPCODE == OP_FileEnd)
        {
            // Se comprueba que el bloque recibido es correcto
            if (checkTramas(n_trama_max, checkFrame) == -1)
            {
                _printf("\n\nError en un bloque recibido. Se solicita reenvio.\n")
                    ;
                enviarTrama(OP_FilePartError, "", 0);
                tramaRecibida = 0;
                goto reenvio;
            }
            else
            {
                // _printf("\n__ACCEPT__\n");
                if (paquete == 0)
```

```
                    _printf("\tLos datos solicitados son: \n\n");

                _printf("%s", txt); // Vuelca por la UART cada bloque de 256 bytes
                paquete++;

                if (OPCODE == OP_FilePartEnd)
                    enviarTrama(OP_FilePartOK, "", 0);
                else if (OPCODE == OP_FileEnd)
                {
                    enviarTrama(OP_Finish, "", 0);
                    _printf("\n\n\tFichero transmitido correctamente \n");
                    tramaRecibida = 0;
                    break;
                }
                tramaRecibida = 0;
                goto reenvio;
            }
        }
        else if (OPCODE == OP_NoFile)
        {
            _printf("Fichero no encontrado\n");
            tramaRecibida = 0;
            STATE = 1;
            break;
        }
        else if (OPCODE == OP_NoInternet)
        {
            enviarTrama(OP_ACK, "", 0);
            _printf("El servidor no ha podido acceder a la direccion WEB\n");
            tramaRecibida = 0;
            STATE = 1;
            break;
        }
        else if (OPCODE == OP_NoCapacity)
        {
            enviarTrama(OP_ACK, "", 0);
            _printf("El fichero solicitado es demasiado grande.\n");
            tramaRecibida = 0;
            STATE = 1;
            break;
        }
        else if (OPCODE == OP_OtherRequest)
        {
            enviarTrama(OP_ACK, "", 0);
            _printf("El servidor ha recibido otra peticion.\n");
            tramaRecibida = 0;
            STATE = 1;
            break;
        }

        enviarTrama(OP_ACK, "", 0);
        tramaRecibida = 0;
    }
fin_timeout:
    return;
}

/**
 * @brief Comprueba si se han recibido todas las tramas correctamente. Si la suma
     de todas las tramas que han llegado es igual
 * a la suma de las tramas teorica suponiendo que el numero de trama mas eelevado
     es correcto, se da por una transmision exitosa.
 *
 * @param finalFrame numero de trama mas alto recibido
 * @param sumFrames suma del numero de tramas que se han ido recibiendo
 * @return 0 si tramas correctas; -1 si tramas erroneas.
```

```
 */
int checkTramas(int finalFrame, int sumFrames)
{
    int sumTramasT = (finalFrame * (finalFrame + 1)) / 2; // Sumatorio del tramas
        cogiendo la ultima recibida.
    // _printf("\nSuma de tramas teorica: %d     Suma de tramas obtenida: %d.\n",
        sumTramasT, sumFrames);
    if (sumTramasT == sumFrames)
        return 0;
    else
        return -1;
}

/**
 * @brief Funcion que se encarga de transmitir la ruta o URL del fichero
     solicitado.
 *
 * @param op Es el codigo de operacion OPCODE
 * @param msg Ruta solicitada
 */
void enviarFilepath(char op, const char msg[])
{
    int ID_final, i, trama;
    CAN_Message Mensaje;

    int n_Frames = _strlen(msg) / 8;          // numero de tramas de 8bytes
    int FinalFrameLength = _strlen(msg) % 8; // longitud de la ultima trama de
        menor longitud
    int n_reenvio = 0, len = 0;
    char op_final;
    int tramasTX;

// _printf("\r\nSe enviarian %d tramas enteras y una trama de %d bytes: \n\r",
    n_Frames, FinalFrameLength);
reenvio2:
    tramaRecibida = 0;
    // Se envian todas las tramas => 32 como maximo => de 0 a 31
    if (FinalFrameLength == 0)
        tramasTX = n_Frames - 1;
    else
        tramasTX = n_Frames;

    for (i = 0; i <= tramasTX; i++)
    // for (i = n_Frames-1; i >= 0; i--)
    {
        trama = i;
        // _printf("%d ", trama);
        // _delay_ms(500);
        // si estamos en la trama 31 se comprueba que OPCODE poner y la longitud
            de la trama.

        if (i == tramasTX)
        {
            // Como se envian paquetes de longitud variada, no siempre se van a
                enviar las tramas completas.
            // En el momento que una trama no se complete, significa que es el
                final del fichero, por lo tanto, enviamos el OP_FilePartEnd.
            if (FinalFrameLength == 0)
                len = 8;
            else
                len = FinalFrameLength;

            txTrama(OP_FilePathEnd, msg, trama, len);
        }
        else
            txTrama(OP_FilePath, msg, trama, 8);
```

```
        if (STATE == 1)
            break; // Si hay fallo de transmision volvemos
    }

    if (OPCODE == OP_Finish)
        _printf("Se ha enviado el filepath: %s \n", msg);
    else if (OPCODE == OP_FilePathError)
    {
        _printf("Ha habido un error al enviar el bloque. Se intenta reenvio.\n");
        n_reenvio++;
        if (n_reenvio > 4)
        {
            _printf("Se ha intentado demasiadas veces. Abortamos\n");
            STATE = 1;
            _delay_ms(2000);
            goto fin_tx;
        }
        goto reenvio2;
    }
    else if (OPCODE == OP_OtherRequest)
        _printf("Ha habido otra petición.\n");

fin_tx:
    return;
}

/**
 * @brief Se encarga de transmitir la trama y espera hasta que recibe el
     asentimiento del receptor. Si tarda mas de 900ms se aborta la transmision y
     se reinicia la comunicacion.
 *
 * @param op OPCODE a transmitir
 * @param msg mensaje a enviar
 * @param trama numero de la trama que se utiliza para enviar la parte del smsg
     correspondiente
 * @param len longitud de la trama
 */
void txTrama(char op, const char msg[], int trama, int len)
{
    int t_init, ID_final;
    CAN_Message Mensaje;

    ID_final = (id_propio << 9) | (op << 5) | trama;
    Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : len, data : {msg[0 +
        (8 * trama)], msg[1 + (8 * trama)], msg[2 + (8 * trama)], msg[3 + (8 *
        trama)], msg[4 + (8 * trama)], msg[5 + (8 * trama)], msg[6 + (8 * trama)],
         msg[7 + (8 * trama)]}};

    CAN_Transmit(&Mensaje);
    if (STATE == 1)
        return; // Si hay fallo de transmision volvemos
    TIMER0_Init();
    while (tramaRecibida == 0)
    {
        if (ms > 900)
        {
            ms = 0;
            sec = 0;
            STATE = 1;
            _printf("Timeout TX\n");
            break;
        }
    } // Si tarda mas de 900ms en responder reintentamos conexion.

    tramaRecibida = 0;
```

```
}

void solicitarFichero(char file[])
{
    int id_Accept;

INIT:
    MCP2515_Write(MCP_CANINTF, 0x00); // Borro notificaciones
    MCP2515_Write(MCP_EFLG, 0x00);    // Borro notificaciones

    // Inicializamos a 0 los ms y los sec, para que no se arrastren valores
        anteriores.
    ms = 0;
    sec = 0;
    TIMER0_Init();

    // Valores iniciales de las siguientes variables de control de flujo.
    OPCODE = 0;
    STATE = 0;
    tramaRecibida = 0;

    enviarTramaRequest(OP_Request, "", 8);
    _printf("\n\n\nSolicito conexion\n");
    if (STATE == 1)
    {
        ms = 0;
        sec = 0;
        _delay_ms(ms_ERROR_TX);
        _printf("Timeout Request\n");
        goto INIT;
    }

    _printf("EFLG: %x TEC: %x REC: %x\n", MCP2515_Read(MCP_EFLG), MCP2515_Read(
        MCP_TEC), MCP2515_Read(MCP_REC));
    _printf("STATUS: %x \n", MCP2515_ReadStatus());

    // Esperamos a recibir una trama con OP_RequestOK. Si pasa mas de sec_ERROR_TO
         se reincia la comunicacion
    while (tramaRecibida == 0 && OPCODE != OP_RequestOk)
    {
        if (sec > sec_ERROR_TO)
        {
            ms = 0;
            sec = 0;
            _printf("No se ha obtenido respuesta\n");
            goto INIT;
        }
    }

    id_Accept = _atoi(txt); // Decodifico el identificador incluido en el mensaje
        de la trama aceptada

    if (id_propio == id_Accept)
        _printf("\nConexion Aceptada\n");
    else
    {
        if (id_Accept == 0)
            _printf("\nEl servidor se encuentra con otro proceso actualmente.\n");
        else
            _printf("\nHa aceptado la peticion con ID: %d.\n", id_Accept);

        _delay_ms(3000);
        goto INIT;
    }

    enviarFilepath(OP_FilePath, file);
```

```
    if (STATE == 1)
    {
        ms = 0;
        sec = 0;
        _delay_ms(ms_ERROR_TX);
        goto INIT;
    }

    GestorRecepcion();
    return;
}
```

5.1.3. Funciones SPI

```
// ================================================================
// ---------------------- COMUNICACION SPI ----------------------
// ================================================================

#include "system.h"


void SPI_Init()
{
    S0SPCR = 0x0020; // xxxx0000001000xxb
                     //               .      BitEnable = 0. 8 bit data
                     //              .       CPHA      = 0. First SCK edge
                     //             .        CPOL      = 0. SCK activa high
                     //            .         MSTR      = 1. SPI MASTER
                     //           .          LSBF      = 0. MSB First
                     //          .           SPIE      = 0. No interrupts
                     //      ....            BITS      = 0000. BitEnable=0, no sense
    S0SPCCR = 30;    // Clock Counter Register
                     // BitRate = PCLK/S0SPCCR
                     // 2e6     = 14745600*4 / S0SPCCR
                     // S0SPCCR = 29.49 -> 30
                     // BitRate = 1966080 bps
}

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

char SPI_EnviaRecibe(char d)
{
    S0SPDR = d;
    while (!(S0SPSR & (1 << 7)))
        ; // Espero a que haya transmitido el dato
    return S0SPDR;
}

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

void SPI_Sel()
{
    IOCLR = CAN_CS;
}

// ----------------------------------------------------------------
void SPI_UnSel()
{
    IOSET = CAN_CS;
}
```

5.2. Código fuente para el ESP8266

5.2.1. Función *setup()* y *loop()*

```
#ifndef __ARDUINO_INO
#define __ARDUINO_INO

#include <Arduino.h>
#include "system.h"
#include "FS.h"

#define baud 115200

#define _printf Serial.printf
#define _delay_ms delay
#define WEB 1

int filtro = 0;
int id_filtro = 1230;
int mk_filtro = 0xFFFFFE00; // Esta mascara acepta todas las tramas que tengan el
    id_filtro
int ide_filtro = 1;

char OPCODE = 1;
int id_propio = 7711;
int id_ajeno;
char id_a[8];
int n_trama;

volatile int STATE = 0;

char txt[1000];
char path[1000] = "/";

volatile int tramaRecibida;

ESP8266WiFiMulti WiFiMulti;

void IRAM_ATTR isr()
{
    tramaRecibida = 1;                // Activo el flag de trama recibida
    escucha();                        // Llamo a la funcion escucha para
        decodificar la trama recibida
    MCP2515_Write(MCP_CANINTF, 0x00); // Borro notificaciones
}

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

    Serial.println("\n\nSERVIDOR INTERFAZ WIFI - CAN\n\n");

    SPI_Init();             // Inicializo el SPI
    CAN_Config(CAN_Speed); // Configuro MCP2515


    _printf("MCP configurado correctamente. \n");

    // Se monta el sistema de ficheros
    bool result = SPIFFS.begin();
    if (result)
        _printf("SPIFFS montado correctamente\n");

    // Realizo la conexion a la red WiFi
    WiFi.mode(WIFI_STA);
```

```
    WiFiMulti.addAP("Manso-WiFi", "davikulo99");
    // WiFiMulti.addAP("OPPO X3 Neo", "davikulo99");
    // WiFiMulti.addAP("Wifi_Lab_Proyectos", "R3BOOTYOURM1ND");

    Serial.println("Conectando...\n");

    if (WiFiMulti.run() == WL_CONNECTED)
        Serial.println("Conectado\n");

    attachInterrupt(4, isr, FALLING); // Activo las interrupciones
}

void loop()
{
inicio:
    OPCODE = 1;
    STATE = 0;
    // Se deshabilita el filtro. Permitimos mensajes de cualquier cliente
    filtro = 0;

    MCP2515_Write(MCP_CANINTF, 0x00); // Borro notificaciones

    CAN_Config(CAN_Speed);
    _printf("\n\nFiltro desactivado\n");
    _printf("\nEscuchando...\n");

    // Espero a que me llegue un request y deshabilito el wdt
    int t_actual = millis();
    while (tramaRecibida == 0 && OPCODE != OP_Request)
    {
        wdt_reset();
        // Si cada 5 segundos no me llega nada reseteo los errores por si acaso.
        if (millis() > (t_actual + 5000))
        {
            CAN_Config(CAN_Speed);
            t_actual = millis();
        }
    }

    _printf("Recibo solicitud de %d\n", id_ajeno);

    // Una vez recibido una solicitud, filtro la recepecion para solo recibir
        tramas con el mismo identificador.
    filtro = 1;
    id_filtro = id_ajeno;
    CAN_Config(CAN_Speed);
    _printf("\nFiltro activado: Mk = %x, Id = %d\n", mk_filtro, id_filtro);

Other_Request:
    tramaRecibida = 0;
    sprintf(id_a, "%d", id_ajeno);
    enviarMsg(OP_RequestOk, id_a, 0);

    GestorRecepcion();
    if (STATE == 1)
    {
        tramaRecibida = 0;
        CAN_Config(CAN_Speed);
        goto inicio;
    }
    else if (STATE == 2)
    {
        tramaRecibida = 0;
        CAN_Config(CAN_Speed);
        goto Other_Request;
    }
```

```
if (strncmp("http", txt, 4) == 0)
{
    NavegarWebPrueba(txt);

    if (STATE == 3) // No ha podido acceder a la pagina web
    {
        tramaRecibida = 0;
        CAN_Config(CAN_Speed);
        WiFi.mode(WIFI_STA);

        Serial.println("Conectando...\n");

        if (WiFiMulti.run() == WL_CONNECTED)
            Serial.println("Conectado\n");
        goto inicio;
    }
}
else
{
    // Lectura del archivo
    strcpy(path, "/");
    strcat(path, txt);

    File file = SPIFFS.open(path, "r");
    if (!file)
    {
        Serial.println("File not found");
        enviarMsg(OP_NoFile, "", 0);
        // Cierre del archivo
        file.close();
    }

    int archivoSize = file.size(); // Tamaño del archivo en bytes
    int blockSize = 1024;          // Tamaño de bloque recomendado para la
        lectura SPIFFS

    // Calcular el número de bloques necesarios para leer todo el archivo
    int numBloques = (archivoSize + blockSize - 1) / blockSize;
    // _printf("n_bloques  = %d\n", numBloques);

    // Leer el contenido del archivo por bloques
    for (int i = 0; i < numBloques; i++)
    {
        char buffer[blockSize] = {0};
        int bytesLeidos = file.readBytes(buffer, blockSize);
        // Serial.write(buffer, bytesLeidos);

        if (i == (numBloques - 1))
        {
            enviarWeb(OP_FileEnd, buffer);
        }
        else
        {
            enviarWeb(OP_File, buffer);
            if (STATE == 1)
            {
                _printf("\nFallo de transmision\n");
                break;
            }
        }
    }

    file.close();
    _printf("Se cierra el fichero\n");
```

```
        goto inicio;
    }
}

#endif
```

5.2.2. Resto de funciones complementarias

```
#include "system.h"

#define _printf Serial.printf
#define _delay_ms delay

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

void enviarMsg(char op, const char msg[], int trama)
{
    int ID_final, i;
    CAN_Message Mensaje;

    String str1 = String(msg);
    int n_Frames = str1.length() / 8;          // numero de tramas de 8bytes
    int FinalFrameLength = str1.length() % 8; // longitud de la ultima trama de
        menor longitud
    int t_init;
    int n_reenvio = 0;
    // _printf("\r\nSe enviarian %d tramas enteras y una trama de %d bytes: \n\r",
         n_Frames, FinalFrameLength);
    switch (op)
    {
    case OP_FilePath:
    case OP_File:
    reenvio:
        for (i = 0; i <= n_Frames; i++)
        // for (i = n_Frames-1; i >= 0; i--)
        {
            trama = i;

            ID_final = (id_propio << 9) | (op << 5) | trama;
            Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : 8, data : {
                msg[0 + (8 * i)], msg[1 + (8 * i)], msg[2 + (8 * i)], msg[3 + (8 *
                 i)], msg[4 + (8 * i)], msg[5 + (8 * i)], msg[6 + (8 * i)], msg[7
                + (8 * i)]}};

            CAN_Transmit(&Mensaje);
            if (STATE == 1)
                break; // Si hay fallo de transmision volvemos

            t_init = millis();
            while (tramaRecibida == 0)
            {
                wdt_reset();
                if (millis() > (t_init + 500))
                {
                    STATE = 1;
                    _printf("Timeout TX\n");
                    break;
                }
            }
            tramaRecibida = 0;
        }

        trama = i;
```

```
        ID_final = (id_propio << 9) | (OP_FileEnd << 5) | trama;
        Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : FinalFrameLength,
             data : {msg[0 + (8 * (trama))], msg[1 + (8 * (trama))], msg[2 + (8 *
            (trama))], msg[3 + (8 * (trama))], msg[4 + (8 * (trama))], msg[5 + (8
            * (trama))], msg[6 + (8 * (trama))], msg[7 + (8 * (trama))]}};
        CAN_Transmit(&Mensaje);
        if (STATE == 1)
            break; // Si hay fallo de transmision volvemos

        t_init = millis();
        while (tramaRecibida == 0)
        {
            wdt_reset();
            if (millis() > (t_init + 500))
            {
                STATE = 1;
                _printf("Timeout TX\n");
                break;
            }
        }
        tramaRecibida = 0;
        if (OPCODE == OP_ACK)
            _printf("Se ha solicitado el fichero con la ruta: %s \n", msg);
        else if (OPCODE == OP_FilePartError)
        {
            _printf("Ha habido un error al enviar la ruta. Se intenta reenvio.\n")
                ;
            n_reenvio++;
            if (n_reenvio > 4)
            {
                _printf("Se ha intentado demasiadas veces. Abortamos\n");
                STATE = 1;
                break;
            }
            goto reenvio;
        }

        break;

    default:
        ID_final = (id_propio << 9) | (op << 5) | trama;
        Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : 8, data : {msg
            [0], msg[1], msg[2], msg[3], msg[4], msg[5], msg[6], msg[7]}};
        CAN_Transmit(&Mensaje);
        if (STATE == 1)
            break; // Si hay fallo de transmision volvemos

        break;
    }
}

/**
 * @brief Se encarga de decodificar la trama recibida asignado el OPCODE y el id
     del transmior.
 */
void escucha()
{
    char SIDH, SIDL, EID8, EID0, DLC, status, remote, type;
    int i;

    if (((status >> 4) & 0x01) == 1)
        type = 1; // Extended identifier
    else
        type = 0; // Standard identifier
    if (((status >> 3) & 0x01) == 1)
        remote = 1; // Standar identifier Remote frame
```

```
    else
        remote = 0; // Extended identifier Remote Data frame

    i = MCP2515_ReadStatus();
    if (i & 0x01) // Buffer 0
    {

        SIDH = MCP2515_Read(MCP_RXB0SIDH);      // Standard identifier high
        SIDL = MCP2515_Read(MCP_RXB0SIDL);      // Standard identifier low
        EID8 = MCP2515_Read(MCP_RXB0EID8);      // Extended identifier high
        EID0 = MCP2515_Read(MCP_RXB0EID0);      // Extended identifier low
        DLC = MCP2515_Read(MCP_RXB0DLC) & 0xF; // Data length code

        n_trama = EID0 & 0b11111;

        for (i = 0; i < DLC; i++)
            txt[i + (8 * n_trama)] = MCP2515_Read(MCP_RXB0D0 + i);
    }
    else // Buffer 1
    {
        SIDH = MCP2515_Read(MCP_RXB1SIDH);      // Standard identifier high
        SIDL = MCP2515_Read(MCP_RXB1SIDL);      // Standard identifier low
        EID8 = MCP2515_Read(MCP_RXB1EID8);      // Extended identifier high
        EID0 = MCP2515_Read(MCP_RXB1EID0);      // Extended identifier low
        DLC = MCP2515_Read(MCP_RXB1DLC) & 0xF; // Data length code

        n_trama = EID0 & 0b11111;

        for (i = 0; i < DLC; i++)
            txt[i + (8 * n_trama)] = MCP2515_Read(MCP_RXB1D0 + i);
    }

    // Obtenemos de la trama el OPCODE y el id del transmisor
    OPCODE = ((EID8 & 0b1) << 3) | (EID0 >> 5);
    id_ajeno = (EID8 >> 1) | ((SIDL & 0b11) << 7) | ((SIDL >> 5) << 9) | (SIDH <<
        12);

    // _printf("%x\n", OPCODE);
}

/**
 * @brief Esta funcion es la encargada de recibir las tramas y comprobar si son
     correctas una vez se haya establecido la conexion con el otro terminal. Una
     vez comprobado que la trama es correcta regresamos a la funcion principal.
 */
void GestorRecepcion()
{
    int checkFrame = 0;
    int n_trama_max = 0;

reenvio:
    memset(txt, '\0', max_MSGlength);
    n_trama = 0;
    checkFrame = 0;
    n_trama_max = 0;
    tramaRecibida = 0;
    STATE = 0;

    int t_actual = millis();
    while (1)
    {
        while (!tramaRecibida)
        {
            wdt_reset();
            if (millis() > (t_actual + 5000))
            {
```

```
                STATE = 1;
                _printf("Timeout\n");
                goto fin_timeout;
            }
        }

        checkFrame += n_trama; // Sumamos el n_trama de las tramas recibidas para
            hacer checksum con las tramas.
        if (n_trama > n_trama_max)
            n_trama_max = n_trama;

        if (OPCODE == OP_Request)
        {
            STATE = 1;
            tramaRecibida = 0;
            _printf("Ha llegado otra peticion con ID: %d\n", id_ajeno);
            STATE = 2;
            break;
        }

        else if (OPCODE == OP_FilePathEnd)
        { // File path completo recibido
            if (checkTramas(n_trama_max, checkFrame) == -1)
            {
                _printf("\n\n__ERROR__\n");
                enviarMsg(OP_FilePathError, "", 0);
                goto reenvio;
            } // Se comprueba
            else
            {
                _printf("\n\n__ACCEPT__\n");
                _printf("Trama recibida es: %s\n", txt);
                enviarMsg(OP_Finish, "", 0);
            }
            // _delay_ms(2000); // Delay para comprobar fallo de transmision en el
                LPC
            tramaRecibida = 0;
            goto fin_timeout;
        }

        else if (OPCODE == OP_FilePartError)
        {
            _printf("Parte del fichero esta corrupto\n");
            tramaRecibida = 0;
            break;
        }

        enviarMsg(OP_ACK, "", 0);
        tramaRecibida = 0;
    }
fin_timeout:
    return;
}

/**
 * @brief Comprueba si se han recibido todas las tramas correctamente. Si la suma
    de todas las tramas que han llegado es igual
 * a la suma de las tramas teorica suponiendo que el numero de trama mas eelevado
    es correcto, se da por una transmision exitosa.
 *
 * @param finalFrame numero de trama mas alto recibido
 * @param sumFrames suma del numero de tramas que se han ido recibiendo
 * @return 0 si tramas correctas; -1 si tramas erroneas.
 */
int checkTramas(int finalFrame, int sumFrames)
{
```

```
    int sumTramasT = (finalFrame * (finalFrame + 1)) / 2; // Sumatorio del tramas
        cogiendo la ultima recibida.
    // _printf("\nSuma de tramas teorica: %d     Suma de tramas obtenida: %d.\n",
        sumTramasT, sumFrames);
    if (sumTramasT == sumFrames)
        return 0;
    else
        return -1;
}

/**
 * @brief Se encarga de enviar el fichero completo, lo divide en bloques y se lo
     pasa a la funcion enviarPkg que se encargara de enviar cada bloque.
 *
 * @param op OPCODE
 * @param msg fichero completo que se quiere transmitir
 */
void enviarWeb(char op, const char msg[])
{
    int i, j;
    char pkg[1000] = {0};

    String str1 = String(msg);
    int n_Pkg = str1.length() / 256;           // numero de bloques de 32 tramas
    int FinalPkgLength = str1.length() % 256; // n. de bytes del ultimo bloque

    // _printf("Se enviarian %d bloques enteros y un bloque de %d bytes: \n\r",
        n_Pkg, FinalPkgLength);
    // Envio los bloques de uno en uno

    STATE = 0;

    for (i = 0; i <= n_Pkg; i++)
    {
        // Si estamos en el ultimo bloque que puede tener un tamaño menor
        // se incluyen solo los bytes necesarios.
        if (i == n_Pkg && op == OP_FileEnd)
        {
            for (j = 0; j < FinalPkgLength; j++)
                pkg[j] = msg[j + (i * 256)];

            enviarPkg(OP_FileEnd, pkg);
            // enviarPkg(OP_File, pkg); // Envio 32 tramas
        }
        else
        {
            // Genero una variable que contenga las 32 tramas
            // y se vaya desplazando a medida que se van enviando
            for (j = 0; j < 256; j++)
                pkg[j] = msg[j + (i * 256)];

            enviarPkg(OP_File, pkg); // Envio 32 tramas
        }

        memset(pkg, '\0', max_MSGlength); // Borro el contenido de pkg despues de
            cada iteracion
        if (STATE == 1)
        {
            _printf("\nTimeout WEB\n");
            break;
        }
    }
    // _printf("Fin WEB\n");
}

/**
```

```
 * @brief Se encarga de enviar un bloque de 256bytes. Comrpueba que se ha
     transmitido correctamente y se puede reenviar 4 veces hasta que se aborta la
     transmision.
 *
 * @param op OPCODE de la trama que se transmitie
 * @param msg la parte del fichero correspondiente al bloque
 */
void enviarPkg(char op, const char msg[])
{
    int ID_final, i, trama;
    CAN_Message Mensaje;

    String str2 = String(msg);
    int n_Frames = str2.length() / 8;          // numero de tramas de 8bytes
    int FinalFrameLength = str2.length() % 8; // longitud de la ultima trama de
        menor longitud
    int n_reenvio = 0, len = 0;
    char op_final;

    // _printf("\r\nSe enviarian %d tramas enteras y una trama de %d bytes: \n\r",
         n_Frames, FinalFrameLength);
    switch (op)
    {
    case OP_File:
    case OP_FileEnd:
    reenvio:
        // Se envian todas las tramas => 32 como maximo => de 0 a 31
        int tramasTX;
        if (FinalFrameLength == 0)
            tramasTX = n_Frames - 1;
        else
            tramasTX = n_Frames;

        for (i = 0; i <= tramasTX; i++)
        // for (i = n_Frames-1; i >= 0; i--)
        {
            trama = i;
            // si estamos en la trama 31 se comprueba que OPCODE poner y la
                longitud de la trama.

            if (i == tramasTX)
            {
                // Como se envian paquetes de 256, siempre se van a enviar las
                    tramas completas.
                // En el momento que una trama no se complete, significa que es el
                     final del fichero, por lo tanto, enviamos el OP_FileEnd.

                if (op == OP_File)
                { // Comprobamos si es fin del bloque o fin de fichero
                    len = 8;
                    op_final = OP_FilePartEnd; // Fin bloque de 256bytes
                }
                else
                {
                    op_final = OP_FileEnd; // Fin de documento
                    // Si coincide que se envian tramas completas en el ultimo
                        bloque
                    // se envian los ultimo 8 bytes del mensaje,
                    if (FinalFrameLength == 0)
                        len = 8;
                    else
                        len = FinalFrameLength;
                }

                txTrama(op_final, msg, trama, len);
            }
```

```
            else
                txTrama(OP_File, msg, trama, 8);
            if (STATE == 1)
            {
                _printf("\nTimeout PKG\n");
                break;
            }
        }

        // if (OPCODE == OP_FilePartOK)
        //     _printf("Se ha enviado el bloque. \n");
        if (OPCODE == OP_Finish)
            _printf("Se ha enviado el fichero completo. \n");
        else if (OPCODE == OP_FilePartError)
        {
            _printf("Ha habido un error al enviar el bloque. Se intenta reenvio.\n
                ");
            n_reenvio++;
            if (n_reenvio > 4)
            {
                _printf("Se ha intentado demasiadas veces. Abortamos\n");
                STATE = 1;
                break;
            }
            goto reenvio;
        }

        break;

    default:
        ID_final = (id_propio << 9) | (op << 5) | trama;
        Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : 8, data : {0, 0,
            0, 0, 0, 0, 0, 0}};
        CAN_Transmit(&Mensaje);
        if (STATE == 1)
            break; // Si hay fallo de transmision volvemos

        break;
    }
}

/**
 * @brief Se encarga de transmitir la trama y espera hasta que recibe el
     asentimiento del receptor. Si tarda mas de 500ms se aborta la transmision y
     se reinicia la comunicacion.
 *
 * @param op OPCODE a transmitir
 * @param msg mensaje a enviar
 * @param trama numero de la trama que se utiliza para enviar la parte del smsg
     correspondiente
 * @param len longitud de la trama
 */
void txTrama(char op, const char msg[], int trama, int len)
{
    int t_init, ID_final;
    CAN_Message Mensaje;

    ID_final = (id_propio << 9) | (op << 5) | trama;
    Mensaje = (CAN_Message){ID : ID_final, rtr : 0, length : len, data : {msg[0 +
        (8 * trama)], msg[1 + (8 * trama)], msg[2 + (8 * trama)], msg[3 + (8 *
        trama)], msg[4 + (8 * trama)], msg[5 + (8 * trama)], msg[6 + (8 * trama)],
         msg[7 + (8 * trama)]}};

    CAN_Transmit(&Mensaje);
    if (STATE == 1)
        return; // Si hay fallo de transmision volvemos
```

```
    t_init = millis();
    while (tramaRecibida == 0)
    {
        wdt_reset();
        if (millis() > (t_init + 1000))
        {
            STATE = 1;
            _printf("\nTimeout TX\n");
            break;
        }
    }
    tramaRecibida = 0;
}
```

5.2.3. Funciones para acceder a internet

```
// ================================================================
// -------------------- FUNCIONES WEB ESP8266 --------------------
// ================================================================
#include "system.h"

void NavegarWebPrueba(String url)
{
    // proporciona una conexión segura utilizando SSL/TLS.
    std::unique_ptr<BearSSL::WiFiClientSecure> client(new BearSSL::
        WiFiClientSecure);

    bool mfln = client->probeMaxFragmentLength("tls.mbed.org", 443, 1024);
    // Serial.printf("\nConnecting to https://www.google.es/\n");
    // Serial.printf("Maximum fragment Length negotiation supported: %s\n", mfln ?
        "yes" : "no");
    // if (mfln)
    // {
    //     client->setBufferSizes(1024, 1024);
    // }

    Serial.print("[HTTPS] begin...\n");

    // Se desactiva la verificación de certificados SSL
    client->setInsecure();

    HTTPClient https;

    if (https.begin(*client, url))
    {

        Serial.print("[HTTPS] GET...\n");
        // start connection and send HTTP header
        int httpCode = https.GET();
        if (httpCode > 0)
        {
            // HTTP header has been send and Server response header has been
                handled
            Serial.printf("[HTTPS] GET... code: %d\n", httpCode);

            // file found at server
            if (httpCode == HTTP_CODE_OK)
            {

                // get length of document (is -1 when Server sends no Content-
                    Length header)
                int len = https.getSize();
```

```
                // read all data from server
                while (https.connected() && (len > 0 || len == -1))
                {
                    // create buffer for read
                    char buff[1024] = {0};

                    // get available data size
                    size_t size = client->available();

                    if (size)
                    {
                        // read up to buff size
                        int c = client->readBytes(buff, ((size > sizeof(buff)) ?
                            sizeof(buff) : size));

                        char *last5 = strrchr(buff, '\0') - 7;

                        if (strstr(buff, "</html>") != NULL)
                        {
                            enviarWeb(OP_FileEnd, buff);
                            goto FINNISH_WEB;
                        }
                        else
                            enviarWeb(OP_File, buff);
                    }
                }
                STATE = 3;
                txTrama(OP_NoInternet, "", 0, 0);
            }
        }
        else
        {
            Serial.printf("[HTTPS] GET... failed, error: %s\n", https.
                errorToString(httpCode).c_str());
            STATE = 3;
            txTrama(OP_NoInternet, "", 0, 0);
        }
FINNISH_WEB:
        https.end();
    }
    else
    {
        Serial.printf("Unable to connect\n");
        STATE = 3;
        txTrama(OP_NoInternet, "", 0, 0);
    }
}
```

5.2.4. Funciones SPI

```
// ================================================================
// ---------------------- COMUNICACION SPI ----------------------
// ================================================================

#include "system.h"

SPISettings SPIS;

// Si se define BB se hace por BitBanging
// Se no se define se hace usando el periférico
#undef BB
// #define BB

//  Definición de pines
```

```
const int SSPin = 5;
const int SCKPin = 14;
const int MISOPin = 12;
const int MOSIPin = 13;

#define _printf Serial.printf

void SPI_Init()
{
#ifdef BB
    pinMode(MISOPin, INPUT);
    pinMode(SSPin, OUTPUT);
    pinMode(SCKPin, OUTPUT);
    pinMode(MOSIPin, OUTPUT);
    digitalWrite(SSPin, HIGH);
#else
    pinMode(SSPin, OUTPUT);
    digitalWrite(SSPin, HIGH);
    SPIS._dataMode = 0;
    SPIS._bitOrder = MSBFIRST;
    SPI.setFrequency(10000000);
    SPI.pins(SCKPin, MISOPin, MOSIPin, SSPin);
    SPI.begin();
#endif
    // _printf("\n\nSPI init\n");
}
// ---------------------------------------------------------------
char SPI_EnviaRecibe(char d)
{
#ifdef BB
    // Por bit-banging
    char _receive = 0;
    for (int i = 0; i < 8; i++) // 8 bits in a byte
    {
        digitalWrite(MOSIPin, bitRead(d, 7 - i)); // Set MOSI
        digitalWrite(SCKPin, HIGH);               // SCK high
        _receive <<= 1;
        _receive = _receive | digitalRead(MISOPin);
        digitalWrite(SCKPin, LOW); // SCK low
    }
    return _receive; // Return the received data
#else
    char _receive = 0;
    _receive = SPI.transfer(d);
    return _receive;
#endif
}
// ---------------------------------------------------------------
void SPI_Sel()
{
#ifdef BB
#else
    SPI.beginTransaction(SPIS);
#endif
    digitalWrite(SSPin, 0);
    // IOCLR = CAN_CS;
}
// ---------------------------------------------------------------
void SPI_UnSel()
{
#ifdef BB
#else
    SPI.endTransaction();
#endif
    digitalWrite(SSPin, 1);
}
```

5.3. Lista de materiales (BOM)

Bill Of Materials for Interfaz_USB_WiFi_CAN

Design Title Interfaz_USB_WiFi_CAN
Author David Manso
Document Number
Revision
Design Created lunes, 17 de abril de 2023
Design Last Modified miércoles, 24 de mayo de 2023
Total Parts In Design 58

9 Capacitors

Quantity	References	Value	PCB Package	Farnell code	Cost
4	C1,C5,C8,C9	100nF	0805	3296132	0,02€
2	C2,C3	3.3uF	0805	2210978	0,21€
1	C4	10nF	0805	1650807	0,19€
2	C6,C7	33pF	0805	1414685	0,055€
Sub-totals:					0,80€

19 Resistors

Quantity	References	Value	PCB Package	Farnell code	Cost
3	R1,R3,R5	1K8	0805	9237526	0,0066 €
3	R2,R4,R6	3K3	0805	1469911	0,0193 €
1	R7	1K	0805	9237496	0,0113 €
2	R8,R9	680R	0805	1653028	0,0193 €
9	R10,R11,R12,R13,R14,R17,R18,R19,R21	10k	0805	3133452	0,142€
1	R20	120R	0805		0,142€
Sub-totals:					1,55€

4 Integrated Circuits

Quantity	References	Value	PCB Package	Farnell code	Cost
1	U1	FT232R	SSOP28	4064605	3,90€
1	U2	ESP-12F	ESP12F	ALIEXPRESS	1,15€
1	U3	MCP2515-I/SO	SO18W	1292239	1,8€
1	U4	MCP2551	SO8	9758569	1,01€
Sub-totals:					7,86€

5 Diodes

Quantity	References	Value	PCB Package	Farnell code	Cost
1	D1	PMEG4010EP	SOD-128	1907688	0,119€
2	D2,D3	S1A	SMA	9843868	0,18€
1	D4	GREEN	LED0805	3796323	0,0658 €
1	D5	RED	LED0805	3796325	0,0608 €
Sub-totals:					0,61€

21 Miscellaneous

Quantity	References	Value	PCB Package	Farnell code	Cost
3	3V3,5V,J4	JUMPER	CONN-SIL2		0,2€
14	CANH,CANL,CS,DTR,GND,MISO,MOSI,POWER,RTS, RXCAN,RXD,SCK,TXCAN,TXD	PIN	PIN		
1	J1	USB-F-TYPE_MINIB	USB-F-TYPE-MINIB	9786465	1,38€
1	J3	MKDS 1,5/ 3-5,08	TBLOCK-I3	3041530	1,46€
1	L1	MMZ2012Y152BT000	0805	2800761	0,0295€
1	X1	QCL20.0000F18B23B	XTAL18	2508461	0,181€
Sub-totals:					3,65€

Totals: 14,46€

miércoles, 24 de mayo de 2023 12:31:53