UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

IMPLEMENTACIÓN DE UNA RED CAN ENTRE MICROCONTROLADORES C2000 Y ESP32 PARA CONTROL Y CONFIGURACIÓN DE DISPOSITIVOS INDUSTRIALES

TRABAJO FIN DE GRADO

Mayo -2025

AUTOR: Sergio Lozano Agulló

DIRECTORES: Roberto Gutiérrez Mazón

Martin Wassermann

RESUMEN

El presente documento se centra en la implementación de un sistema de comunicación basado en el protocolo CAN bus entre dos microcontroladores diferentes, cada uno encargado de controlar un dispositivo distinto. El objetivo principal fue lograr una comunicación eficiente y confiable que permitiera el intercambio de información entre ambos sistemas, como paso inicial a construir una red de múltiples dispositivos controlables por el usuario desde uno de ellos. Para ello, se abordaron retos de diseño y configuración en base a las características del bus CAN. Se desarrollaron programas específicos garantizando que cada microcontrolador pudiera interpretar correctamente la información recibida del otro. Por último, se incorporó un ejemplo práctico en el que el usuario puede participar activamente en la comunicación.

Los resultados demostraron que, con un adecuado ajuste de parámetros, es posible establecer una comunicación robusta entre ambos microcontroladores, permitiendo así la creación de una red de múltiples dispositivos que interactúen entre ellos, como en una aplicación industrial.

Palabras clave: CAN bus, ESP32, C++, microcontrolador.

AGRADECIMIENTOS

En primer lugar, a mi familia, que me han dado la posibilidad de desarrollar mi carrera académica y me han apoyado en todo momento.

En segundo lugar, a la empresa en la que he realizado las prácticas y este Trabajo de final de grado, Wagner Magnete, por ayudarme a dar mis primeros pasos en el ámbito profesional y por ofrecerme la posibilidad de seguir aprendiendo. Sin ellos este trabajo no hubiera sido posible. En especial quiero agradecer a Martin, mi tutor en la empresa, por ayudarme a encontrar un tema interesante y por guiarme a lo largo de estos meses de esfuerzo.

También quiero agradecer a mi tutor académico Roberto Gutiérrez Mazón, por haberme guiado y ayudado en la presentación de la memoria, así como a todos los profesores del grado por su labor.

Por último, quiero agradecer a todos mis amigos y aquellas personas con las que me he cruzado estos años por haber formado parte de esta etapa en mi vida.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

1.1 INTRODUCCIÓN

El sector industrial está compuesto por infinidad de máquinas que facilitan o realizan el trabajo y ponen en práctica la alta capacidad tecnológica de hoy en día. Cada una de estas máquinas está compuesta por subsistemas que se encargan de tareas más específicas, pero que en conjunto tienen el objetivo de transformar los productos o servicios desde el inicio como materia prima hasta el producto acabado. Y cada uno de estos subsistemas contiene en su interior un "cerebro" que manda órdenes a sensores y actuadores. Un "cerebro" formado por componentes electrónicos, como por ejemplo un ordenador, un autómata o en el caso de este proyecto: un microcontrolador.

Un microcontrolador es un circuito integrado que combina un microprocesador, memoria, puertos de entrada/salida y otros componentes con la finalidad de controlar y ejecutar tareas específicas, con mínima o ninguna intervención humana. Se trata de una microcomputadora diseñada para aplicaciones embebidas, tales como dispositivos móviles, electrodomésticos o vehículos.

La empresa Wagner Magnete GmbH. & Co. KG [1] está especializada en la producción de electroimanes de varios tipos, y también fabrica dispositivos para el control de los mismos.

Este proyecto se centra uno de estos controladores de imanes, el USG758, cuya función principal es magnetizar y desmagnetizar un imán, mediante programas ajustables según la necesidad específica para obtener el mejor resultado. Este dispositivo es capaz de controlar la electricidad que circula por el imán de forma precisa, variando así su campo magnético. Son requeridos en especial en aplicaciones en las que la imantación remanente o residual de los electroimanes suponga un problema, por ejemplo, en dispositivos de sujeción para el mecanizado de piezas metálicas grandes, donde la imantación remanente puede ser muy fuerte e impedir que las piezas se separen, aun cuando el imán está inactivo.

Figura 1.1. Dispositivo USG758.

El controlador USG758, mostrado en la Figura 1.1, es fabricado por Wagner Magnete y se compone principalmente de una placa electrónica controlada por un microcontrolador, el TMS320F28069. La placa actual se encarga del control de los imanes mediante electrónica de potencia y sensores, y mediante la placa frontal se ofrece una interfaz simple al usuario, para revisar el estado del imán o seleccionar funcionalidades diferentes para desimantar adecuadamente, entre otras cosas. Desde la placa frontal se puede interactuar con el imán mediante los botones y la pantalla, o también mediante un autómata a través de los puertos de la izquierda. Cada puerto para el autómata hace referencia a una variable o instrucción que puede estar con valor activo o inactivo. Entonces se puede programar un autómata para que controle el comportamiento o rutinas de uno o varios dispositivos. La instalación frecuentemente consiste en un armario cerrado con varios dispositivos USG758, que forman parte de un sistema más complejo, por ello la utilidad de controlarlos con un autómata.

Para una futura versión del producto se ha pensado, entre otras cosas, en crear una consola o terminal como la de la Figura 1.2, que sea equivalente a la placa frontal del USG758, pero con menos funcionalidades a cambio de una mayor portabilidad y manejo. El usuario podrá controlar el dispositivo USG758 desde la consola. Esta consola se comunicará con

el dispositivo USG758 mediante cable a una distancia generalmente menor a 10 metros, aunque no se descarta una separación mayor en algún caso. La electrónica de la consola será controlada por un microcontrolador ESP32, debido a su versatilidad, bajo coste y cantidad de ejemplos disponibles. En un producto anterior de la empresa ya había una consola de aspecto similar, aunque mucho más simple. La antigua consola trataba de simular las órdenes de un autómata, ya que cada botón o elemento de la consola iba conectado mediante un cable diferente a uno de los puertos para el autómata. Pero esto era muy engorroso debido a la cantidad de cables y su conexión.

Figura 1.2. Modelo 3D de la consola

La motivación del proyecto trata en implementar este sistema de comunicación entre la consola y el controlador del imán, y llevarla a cabo con los objetivos propuestos, aplicando algunos de los conocimientos adquiridos en el grado y adquiriendo un conocimiento más profundo en el mundo de los microcontroladores y el CAN bus.

1.2 OBJETIVOS

El objetivo principal del proyecto es implementar una red de comunicación CAN entre el controlador del imán (TMS320F28069) y la consola (ESP32). En la red CAN se intercambiarán datos en tiempo real para mostrar y gestionar el funcionamiento del dispositivo USG758 desde la consola.

Otros objetivos del trabajo son:

- Gestionar el envío y recepción de los datos transmitidos adecuadamente.

- Creación de pequeños ejemplos para que el usuario interactúe con la consola.
- Implementar la comunicación de forma que sea compatible con el funcionamiento normal del USG758.
- Analizar que no haya errores en la comunicación.
- Posibilidad de comunicar la consola con varios dispositivos USG.

1.3 JUSTIFICACIÓN DEL MÉTODO SELECCIONADO. CAN BUS.

El protocolo de comunicación que hemos elegido es el CAN. El bus CAN nos permite añadir o quitar nodos al bus libremente y admite tasas de transmisión de datos relativamente altas. La topología de bus significa que todos los dispositivos comparten el mismo canal de comunicación. Gracias a que requiere de únicamente dos cables y a su topología de bus, se pueden reducir los costes en cables y conectores. Como fue pensado para entornos industriales o sistemas críticos, como es el caso de los coches, el protocolo se ha diseñado con una robustez muy alta gracias a la comunicación diferencial y a los mecanismos propios del bus CAN, como la gestión de errores o arbitraje.

Pero en general, el bus CAN no es el protocolo de comunicación más usado entre microcontroladores, ya que, aunque todo suena muy bien, puede ser difícil de implementar en algunos casos, además de que cada nodo requiere de un controlador específico de CAN y su transceptor. La Tabla 1, a continuación, muestra algunas características de los protocolos de comunicación más usados en microcontroladores. Así podemos comparar fácilmente algunas ventajas o desventajas de CAN frente a otros protocolos.

	I2C	SPI	RS485	CAN
Velocidad de transmisión máx	5 Mbit/s	~50 Mbit/s	10 Mbit/s	1 Mbit/s
Máx. de nodos	10-20	Depende del nº de pines (5-10)	32	100
Tipo de comunicación	Síncrono	Síncrono	Asíncrono, half duplex	Asíncrono, half duplex
Longitud de cable máxima	1m	20cm	1200m (a 100kbit/s)	1000m (a 50kbit/s)
Nº de cables	2	4	2	2
Robustez	Muy poco robusto	Poco robusto	Robusto	Muy robusto
Topología física	Maestro-Esclavo	Maestro-Esclavo	Maestro-Esclavo, bus	Multimaster, bus

Tabla 1. Tabla comparativa de diferentes buses de comunicación

Viendo la comparativa de los diferentes buses de comunicación populares para la comunicación entre microcontroladores o componentes electrónicos, podemos descartar la comunicación SPI y I2C debido a su longitud, que se queda corta para nuestra aplicación. Aunque en SPI realmente se podría ampliar esta longitud en sacrificio de una tasa de transmisión de datos mucho menor, aun así, lo vamos a descartar porque también requeriría de muchos pines o puertos en el nodo maestro si queremos conectar muchos dispositivos. Por otro lado, el estándar RS485 y CAN son de características muy similares, aunque en algunas como la velocidad de transmisión es mucho superior el RS485. Ambos nos ofrecen una comunicación en un bus mediante dos cables, mucha longitud y una cantidad suficiente de nodos. Pero nos hemos decantado por el CAN bus debido a que el protocolo CAN incluye sistemas para la detección y manejo de errores o el arbitraje, que hacen la comunicación mucho más robusta. El sistema multimaestro también puede ser interesante si en el futuro se añaden más dispositivos diferentes a la red.

Todas estas cualidades nos han llevado a decidirnos por el CAN bus como el más óptimo para nuestro proyecto.

2 MATERIAL Y METODOS

2.1 PROTOCOLO CAN

El protocolo CAN, por sus siglas “Controller Area Network”, se trata de un protocolo de comunicación en serie ideado para el transporte de mensajes o datos en una red de múltiples dispositivos. Se caracteriza por ser un protocolo con nodos multimaestro, es decir, cualquier dispositivo puede ser un nodo emisor o receptor. Cuenta con tasas de transmisión de datos de hasta 1Mbit/s. A día de hoy es uno de los protocolos de comunicación más ampliamente usados en el interior de los automóviles, aunque también se utiliza en otros lugares como fábricas o hospitales.

La tecnología CAN fue diseñada por Bosch en 1986 como resultado de la demanda de conectar cada vez más unidades de control electrónicos (ECUs) y sensores en el interior de los coches [2].

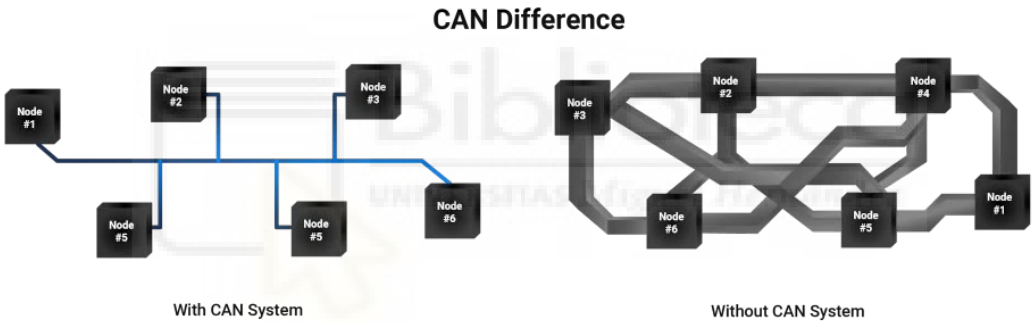

Figura 2.1. Conexión entre nodos con y sin CAN bus [3]

Los fabricantes de automóviles se encontraban en problemas con el elevado número de cables que se usaban, hasta que el CAN bus lo resolvió, ya que se establece un mismo bus compartido por todos los dispositivos, como se muestra en la Figura 2.1, y la conexión física de éste se realiza únicamente mediante dos cables.

Las ventajas más importantes del bus CAN son:

- Capacidad de filtrar solo los mensajes deseados. También se pueden detectar errores y priorizar mensajes según su importancia.
- La red es robusta debido a que un nodo con fallos de transmisión no puede obstruir la comunicación, se desconecta automáticamente en ese caso. La red también es resistente a interferencias.

- Sencillez y bajo coste debido al tipo de conexión.
- Es una red muy escalable ya que se pueden añadir o eliminar nodos sin importar su orden, y el límite de nodos es muy alto, alrededor de 100 nodos.
- Cuenta con un sistema de arbitraje, un sistema sincronización de bits y un CRC (Cyclic Redundancy Check) que comprueba que los datos del mensaje sean correctos.

Según el modelo OSI (Open Systems Interconnection), que se usa como modelo de referencia para dividir el protocolo de una red en hasta 7 capas jerarquizadas según sus funciones propias, el protocolo CAN solo especifica las dos primeras capas: la capa física y la capa de enlace de datos. Las características específicas de estas dos capas están estandarizadas por la norma ISO 11898-1. [2]

2.1.1 CAPA FÍSICA

La comunicación CAN se realiza mediante dos cables trenzados, CANH (CAN High) y CANL (CAN Low). Se le llama comunicación diferencial porque el receptor interpreta la resta entre estas dos señales. Esta resta o diferencia entre las dos líneas elimina las interferencias que afecten en ambos cables. La señal de CANL es inversa a CANH, o viceversa. Como se observa en la Figura 2.2, la señal diferencial se puede interpretar como dos estados:

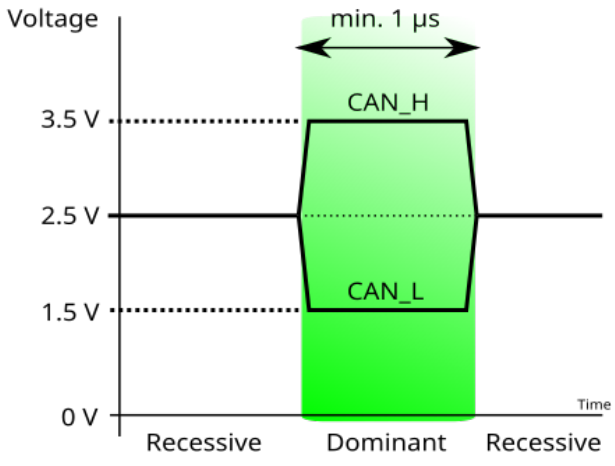

Figura 2.2. Estados del bus CAN. [4]

- Estado Dominante: equivale a un 0. CANH tiene un voltaje mayor (normalmente 3,5V) y CANL un voltaje menor (normalmente 1,5V).

- Estado recesivo: equivale a un 1. CANH y CANL tienen el mismo voltaje (normalmente 2,5V).

Es muy importante saber que el estado dominante se sobrepone al estado recesivo, por tanto, el ‘0’ se sobrepone al ‘1’. Podemos usar la Figura 2.3 como apoyo visual:

'0' = dominante o comunica, '1' = recesivo o no comunica			
Nodo 1	1	0	1
Nodo 2	0	0	1
Nodo 3	1	0	1
Estado del bus	0	0	1

Figura 2.3. Estado del bus CAN

Si al menos uno de los nodos transmite un mensaje, el bus estará en estado dominante (0). Si ningún nodo transmite, el bus se encuentra en estado recesivo (1). Este sería el comportamiento equivalente al de una puerta lógica AND.

Como se muestra en la Figura 2.4, una red CAN está formada por:

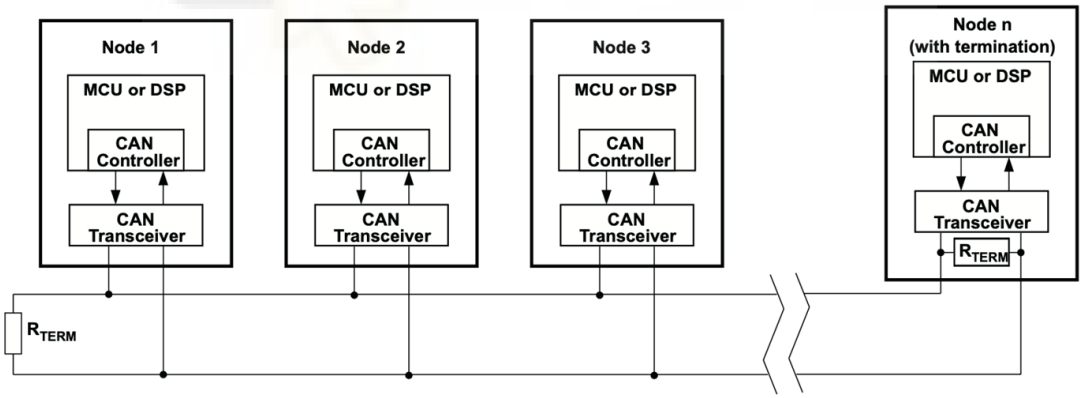

Figura 2.4. Ejemplo de conexión de una red CAN. [5]

- Dos o más nodos: cada nodo contiene un controlador CAN, que gestiona los datos y mensajes, y un transductor, que transmite y convierte los mensajes por el bus.
- Dos cables: uno de ellos asignado a CANH (CAN High) y otro a CANL (CAN Low).

- Dos resistencias de terminación de 120 Ω: estas resistencias eliminan la reflexión de la señal.

Todos estos componentes de la red se disponen formando una topología física de tipo bus, que se define por tener un solo canal semidúplex al cual se conectan todos los nodos. La comunicación semidúplex o "Half-Duplex" se trata simplemente de que cada nodo puede enviar o recibir datos, pero no ambas a la vez. Y todos estos datos se transmiten a través de una comunicación de tipo serie, es decir, los bits se transmiten de uno en uno.

Además, el protocolo cuenta con un sistema de sincronización de bits. Esto es necesario para corregir pequeños desajustes de tiempo entre los nodos. No usa ningún mecanismo de sincronización al inicio o fin de una conexión, sino que cuando cada controlador se conecta a la red escucha los mensajes enviados y con esto ya es capaz de sincronizarse. Un mensaje (o trama) está compuesto de bits y cada bit está a su vez compuesto de 4 intervalos de tiempo, medidos en "quantums", que son unidades de tiempo muy pequeñas. El controlador tiene la capacidad de modificar la duración de algunos de estos intervalos para sincronizar su reloj interno correctamente y leer los bits.

Se toma de referencia el flanco de la señal, es decir, cuando cambia de estado. Resumidamente, el Bit Time o tiempo de bit tiene estas 4 fases para sincronizarse adecuadamente:

- La fase de sincronización: ocurre la transición de un bit a otro y tiene una duración fija de 1 "quantum".
- La fase de propagación: compensa los retrasos debido a la distancia física que recorre la señal.
- Las fases 1 y 2: su duración puede ser modificada entre 1 y 8 "quantums" para que el controlador CAN se sincronice.

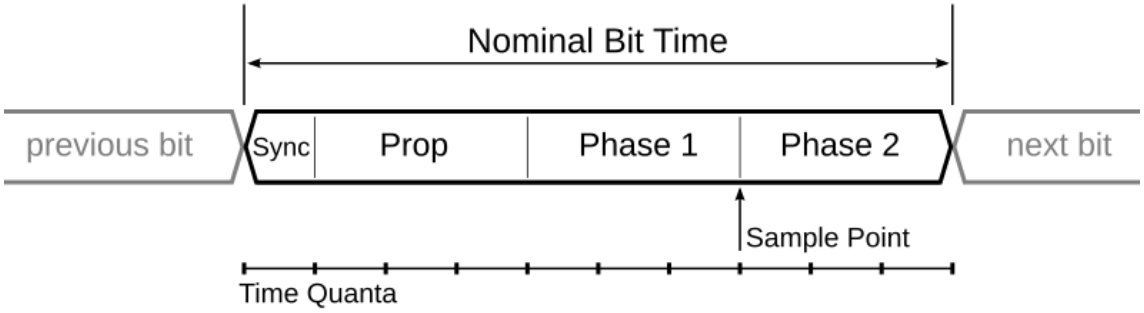

Figura 2.5. Intervalos internos del Bit Time.

En todos los controladores deben ser ajustados estos intervalos mostrados en la Figura 2.5 para configurar el Baud Rate o tasa de transmisión de bits. El cálculo de estos parámetros se explica en profundidad en el artículo de Florian Hartwich y Armin Bassemir "The Configuration of the CAN Bit Timing" [6]. En internet hay varias páginas web que calculan la configuración concreta del Bit Time en función del tipo de controlador, velocidad del reloj interno, Baud Rate deseado y otros parámetros más específicos. Esto es de gran ayuda, pues, aunque los cálculos no sean muy complejos, son diferentes para cada tipo de controlador.

2.1.2 CAPA DE ENLACE DE DATOS

Antes de explicar cómo funciona el arbitraje y la detección de errores, vamos a explicar las tramas o tipos de mensaje que se usan en CAN. El protocolo CAN establece que, en funcionamiento normal, las tramas o mensajes dejarán un espacio al final de tres bits recesivos ('1's), antes de empezar una nueva trama. También, por norma y para asegurar la sincronización, se emplea un relleno de bits, o Bit Stuffing: cada 5 bits iguales consecutivos, se introduce un bit opuesto, pero no tiene ningún efecto en la comunicación, ya que el receptor también lo detecta como bit de relleno. Explicado esto, las diferentes tramas son:

1. Trama de Datos (Data Frame):

   La trama de datos es la más usada, ya que se utiliza para enviar mensajes. Según el tamaño del Identificador dentro del campo de arbitraje (o "Arbitration Field"), podemos distinguir dos tipos:

   - CAN Estándar (CAN 2.0A): con identificador de 11 bits.
   - CAN Extendido (CAN 2.0B): con identificador de 29 bits.

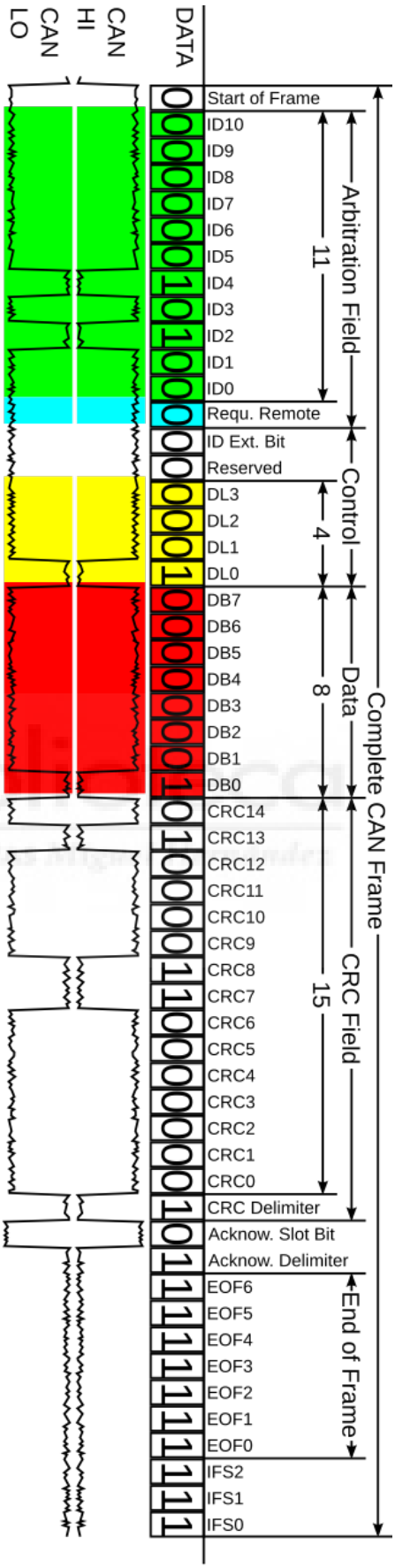

Figura 2.6. Trama CAN de datos estándar. [4]

El formato de la trama de datos estándar, mostrado en la Figura 2.6, se detalla en la Tabla 2 a continuación:

Nombre del campo	Longitud (bits)	Finalidad
Inicio de trama	1	Demarca el comienzo de una transmisión.
Identificador - ID	11	Un identificador (único) que también representa la prioridad de la trama.
Petición de transmisión remota - RTR	1	Dominante (0) para tramas de datos y recesivo (1) para tramas de peticiones remotas.
Bit de extensión de identificador - IDE	1	Dominante (0) para el formato base (identificador de 11 bits).
Bit reservado (r0)	1	Bit reservado. Debe ser dominante (0), pero aceptado tanto dominante como recesivo.
Código de longitud de datos - DLC	4	Número de bytes de datos en el mensaje, entre 0 y 8. Si este campo es mayor que 8 el mensaje será de 8 bytes como máximo de cualquier modo.
Campo de datos	0–64 (0-8 bytes)	Datos de la trama (la longitud del campo viene dada por el código de longitud de datos o DLC).
CRC	15	Verificación por redundancia cíclica. Código que verifica que los datos fueron transmitidos correctamente.
Delimitador CRC	1	Debe ser recesivo (1).
Hueco de acuse de recibo - ACK	1	El transmisor emite recesivo (1) y cualquier receptor emite dominante (0).
Delimitador ACK	1	Debe ser recesivo (1).
Fin de trama EOF	7	Debe ser recesivo (1).

Tabla 2. Bits de trama de datos estándar. [4]

2. Trama Remota:

   La trama remota es igual a la trama de datos, pero suprimiendo el campo de datos y ajustando el bit RTR como recesivo, que indica de que se trata de una trama remota. Por lo general, se usan muy poco, ya que la utilidad principal que tienen es para implementar un sistema de Petición-Respuesta, es decir, un nodo envía una trama remota a otro nodo, y cuando se recibe, el nodo que recibió la trama reacciona con una respuesta. Aunque en realidad una trama de datos puede usarse con el mismo propósito.

3. Trama de Error:

   La trama de error tiene un formato completamente distinto al de otras tramas, ya que se compone de 6 bits con el mismo estado consecutivos (ampliable 6 bits más) para indicar que hay un error, seguido de 8 bits de terminación en estado recesivo. Este formato diferente viola las reglas del CAN bus, como el relleno de bits y el espaciado final, y por eso señaliza a los nodos que se produjo un error. La trama de error funciona de esta forma:

   Cuando un nodo detecta un error en el bus, este envía inmediatamente una trama de error que detectan los demás nodos. Cuando los nodos detectan esta trama de error (si no han detectado el error previamente), la repiten transmitiendo una trama de error a la vez (por esta repetición es ampliable 6 bits más la trama de error, hasta 12 bits).

   Existen dos tipos de errores: Error Activo, 6 bits dominantes (0); y Error Pasivo, 6 bits recesivos (1). Se puede profundizar más sobre el comportamiento del bus frente a cada error dependiendo si el error se detecta por un nodo transmisor o receptor, o si es un error activo o pasivo. En la bibliografía se puede encontrar una página que explica exactamente esto [7]. Además, los controladores CAN de cada nodo tienen un elaborado sistema de contadores encargados de no saturar el bus con tramas de error.

4. Trama de sobrecarga:

   La trama de sobrecarga es muy poco usada, ya que la mayoría de controladores CAN son capaces de gestionarlo con otros métodos. Tiene un formato similar a la trama de error y se envía cuando el nodo está muy ocupado, indicando a los demás nodos que dejen más espacio entre el envío de tramas. También viola las reglas del estándar CAN como el relleno de bits o espaciado final.

Una vez explicadas los diferentes tipos de tramas, podemos definir cómo funciona el arbitraje.

El arbitraje o acceso al medio es el protocolo o instrucciones que deben seguir los nodos para decidir quién obtiene acceso al bus, en caso de que dos o más nodos quieran transmitir un mensaje simultáneamente. Este proceso de arbitraje sucede en el campo de arbitraje ("Arbitration Field"), que comprende los bits para el Identificador y el bit RTR.

Supongamos que el bus está vacío y de repente dos nodos empiezan a transmitir a la vez. Los nodos observan la señal en el bus mientras transmiten el campo de arbitraje. Si durante el envío del campo de arbitraje un nodo detecta un estado dominante (0) en el bus cuando está enviando un estado recesivo (1), este deja de transmitir y se convierte en un receptor. Esto se traduce en que el identificador que tenga antes un '1', es decir, un identificador mayor, pasará a ser un receptor. Por tanto, el transmisor con identificador más pequeño (recordemos que el '0' es dominante) tiene mayor prioridad. Los nodos que pierdan en el proceso de arbitraje intentarán enviar el mensaje tras acabar la transmisión actual. Con este mecanismo se evita la colisión de mensajes.

El protocolo CAN cuenta también con un sistema de detección de errores. Para ello, cada controlador CAN cuenta con un contador para errores de transmisión y otro para errores de recepción. Según el número de los contadores, se distinguen tres estados:

- Error Activo: el nodo funciona en estado normal y solo puede enviar tramas de error activas, cortando el tráfico por el bus.
- Error Pasivo: el nodo está más limitado, pero todavía tiene acceso al bus. Si detecta errores, no cortará el tráfico del bus, porque solo transmite errores pasivos.
- Bus-Off o anulado: si el contador detecta un número de errores excesivo, este desconecta el transceptor del bus y se aísla de la comunicación.

Ahora ya sabemos aproximadamente cómo funciona la detección de errores, pero no se ha mencionado nada de que tipos de error hay. Los 5 tipos de errores son:

1. Bit Error: sucede si un nodo transmisor detecta en el bus un bit diferente al que envió. Recordemos que los nodos monitorizan el bus mientras transmiten.

2. “Bit Stuffing Error”, o error de relleno de bits: se activa si se rompe la regla del relleno de bits, que dice que, tras 5 bits iguales consecutivos, se introducirá un bit opuesto. Sucede, por ejemplo, si hubiera 6 bits consecutivos iguales.
3. “Frame Error”, o error de campo: sucede si un campo que debería tener un valor fijo, como el bit delimitador de CRC o ACK que deben tener siempre el valor ‘1’, tiene un valor no correspondiente.
4. ACK error, o error de reconocimiento: cuando un nodo envía un mensaje, el bit de reconocimiento, o ACK bit, es transmitido con valor recesivo, y el nodo o nodos receptores pueden responder en este momento un bit dominante, para verificar que el mensaje ha sido reconocido. Si este bit de reconocimiento no es dominante, es decir, nadie recibió el mensaje, se activa el ACK error.
5. CRC Error: cuando un nodo envía un mensaje, a partir de este se calcula el campo CRC con un algoritmo, y el nodo emisor añade este CRC al mensaje. Luego, cuando otro nodo recibe el mensaje, vuelve a calcular el CRC para verificar que el mensaje ha llegado correctamente. Si el CRC calculado por el emisor es diferente al del calculado por el receptor, se activa el CRC Error.

Cada error aumenta el contador de error de transmisión o recepción en un número determinado según el tipo de error, y por cada mensaje enviado o recibido correctamente, decrementa el contador en uno.

Con todo esto ya tenemos una base sólida para entender el funcionamiento y componentes de un bus CAN.

2.2 MATERIAL EMPLEADO

2.2.1 HARDWARE

2.2.1.1 ESP32

El ESP32 (ESP32-WROOM-32E concretamente) es una serie de microcontroladores SoC (System on Chip) versátil, altamente integrado y de bajo consumo desarrollado por Espressif Systems. Está diseñado para aplicaciones de Internet de las cosas (IoT), domótica, redes industriales y comunicaciones inalámbricas, entre otros usos. Con su poderosa arquitectura, conectividad inalámbrica avanzada y numerosas interfaces periféricas, ofrece una solución ideal para una amplia gama de aplicaciones. Sus características principales son:

- CPU: Microprocesador Xtensa LX6 Dual-Core 32-bit con frecuencia de hasta 240MHz
- Conectividad Wi-Fi y Bluetooth avanzadas
- Memoria SRAM de 520 KB y flash externa de hasta 4 MB
- 34 GPIOs (Entrada/Salida de Propósito General) programables
- 18 canales ADC (Analog-Digital Converter) y 2 canales DAC (Digital-Analog Converter)
- Interfaces con protocolos de comunicación como I2C, SPI, UART, y el que nos interesa: CAN, que en ocasiones aparece como TWAI (Two-Wire Automotive Interface)
- Modulación PWM
- Voltaje de funcionamiento: 3.3V

En la Figura 2.7 se puede observar todas las características que contiene el chip ESP32:

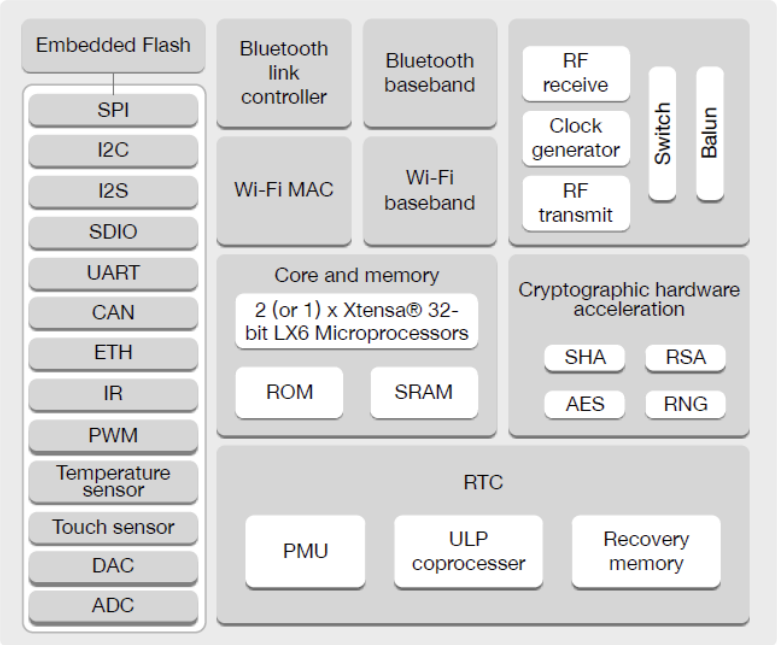

Figura 2.7. Diagrama de bloques funcionales del ESP32 [8]

El controlador CAN integrado en el ESP32 es "equivalente" al controlador CAN SJA1000 y es compatible con CAN 2.0 y tiene tres modos de operación: normal, solo escucha y auto testeo. Admite una tasa de bits o baud rate desde 25kBit/s hasta 1Mbit/s. Tiene un buffer FIFO de 64 bytes para los mensajes recibidos (3-13 bytes por mensaje), y permite el uso de hasta dos filtros de aceptación. Cuenta con detección y manejo de errores.

2.2.1.2 TMS320F28069

El TMS320F28069 es un microcontrolador perteneciente a la familia de microcontroladores C2000™ de 32-bit, fabricados por Texas Instruments. Esta familia de microcontroladores está diseñada para aplicaciones de control en tiempo real en la industria. Algunas de sus características son:

- CPU: C28x con frecuencia de hasta 90 MHz
- Memoria RAM de 100 KB y flash de 256 KB
- 54 GPIOs (Entrada/Salida de Propósito General) programables
- Interfaces con protocolos de comunicación como I2C, SPI, UART y CAN
- Modulación PWM
- Voltaje de funcionamiento: 3.3V

En la Figura 2.8 se puede observar todas las características que contiene el chip ESP32:

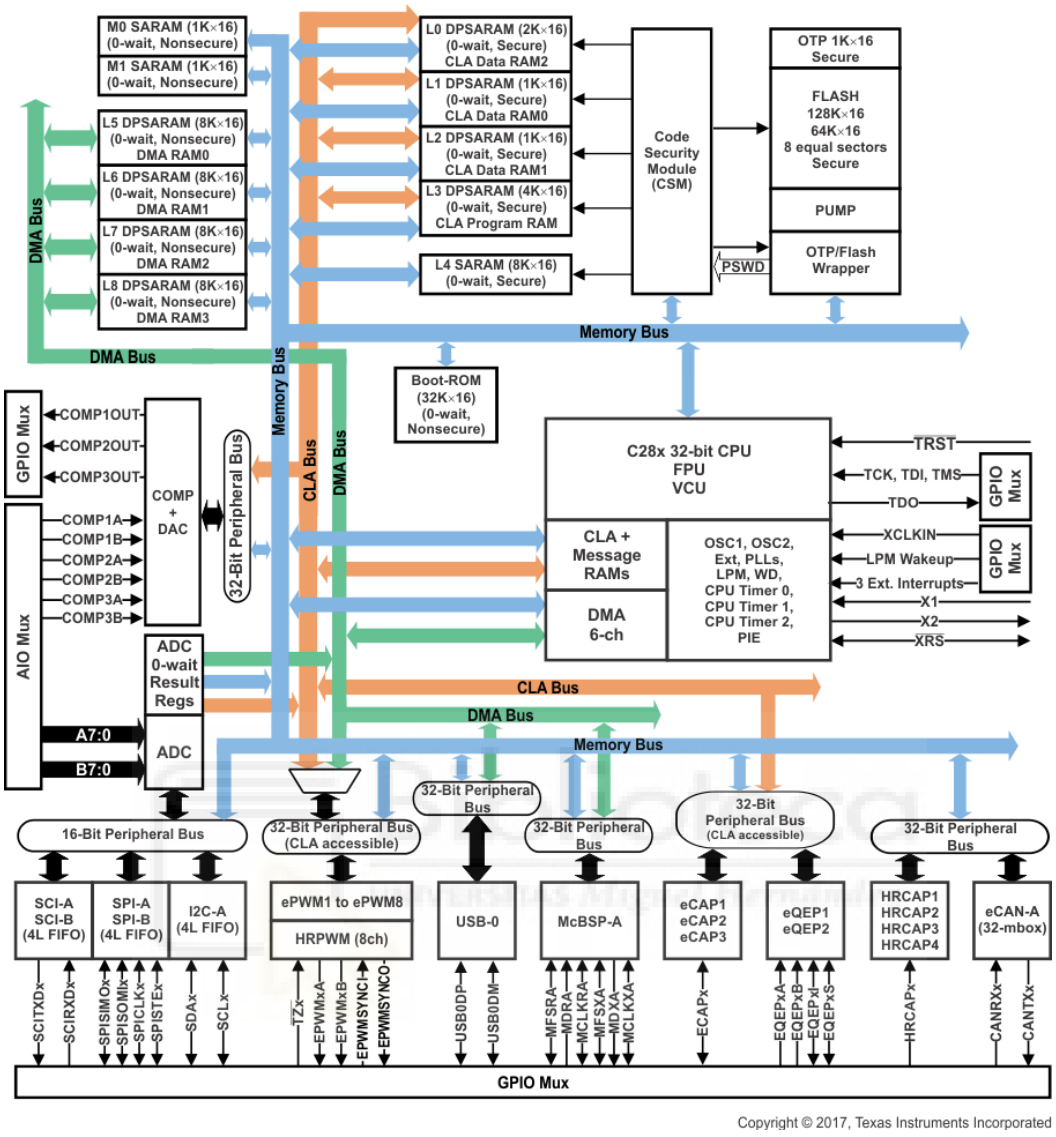

Figura 2.8. Diagrama de bloques funcionales del TMS320F28069 [9].

Para programarlo usaremos un depurador XDS110 Debug Probe, aunque no es estrictamente necesario.

El TMS320F28069 tiene un controlador CAN con dos módulos de memoria, eCAN-A y eCAN-B. Cada uno tiene dos segmentos de memoria dedicada, una a registros de control o estatus y otra al acceso a los buzones o mailboxes (MBOX). El módulo eCAN-A se puede observar en la Figura 2.9.

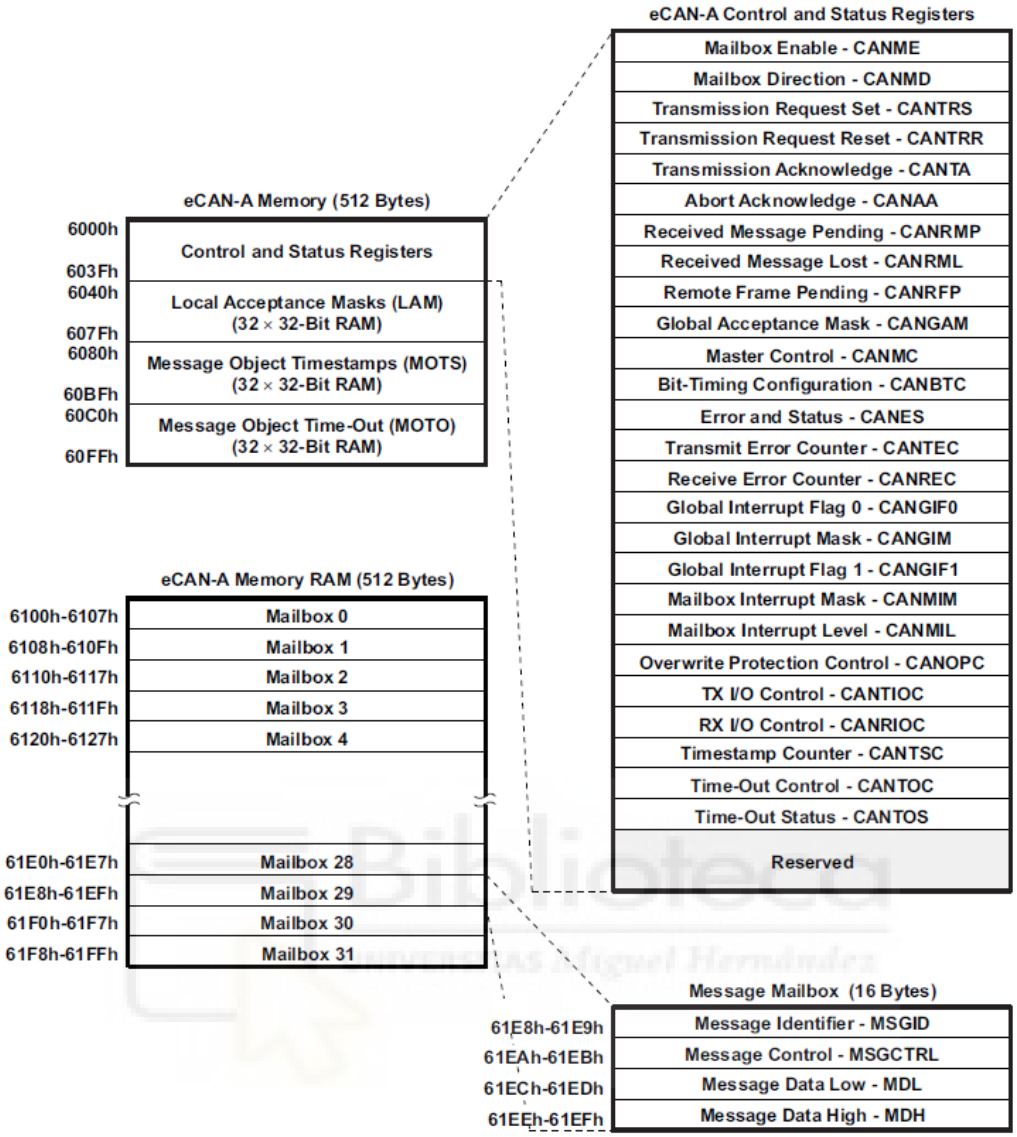

Figura 2.9. Mapa de memoria del módulo eCAN-A [10]

Del primer apartado de memoria, el cuál son los registros de control y estatus, vamos a nombrar los que hemos utilizado y su función:

- CANME: habilita los Mailboxes para que puedan funcionar.
- CANMD: configura los MBOXs como transmisores o receptores.
- CANTRS: activa el proceso de transmitir un mensaje desde el MBOX transmisor seleccionado.
- CANTA: este registro se activa automáticamente cuando el mensaje transmitido es recibido por otro nodo correctamente.
- CANRMP: se activa automáticamente cuando hay un mensaje recibido listo para procesarse.

- CANMC: contiene varios ajustes del módulo CAN.
- CANES: contiene información sobre los registros de error y estatus.
- CANTIOC y CANRIOC: estos registros configuran los puertos del microcontrolador como CANTX y CANRX.

Los apartados de memoria restantes son MOTS, MOTO y LAM. MOTS y MOTO están pensados para gestionar estampas de tiempo y temporizadores. LAM es para configurar la máscara o filtro de aceptación de cada Mailbox. De estos tres apartados de memoria solo hemos usado LAM en el proyecto.

En la memoria RAM se almacenan los 32 MBOXs, cada uno con cuatro capas de memoria:

- MSGID: Almacena el identificador y otros bits de control del mailbox.
- MSGCTRL: Almacena la configuración para tramas RTR, la prioridad de transmisión y el número de bytes a transmitir en el mensaje.
- MDL: almacena 4 bytes de datos.
- MDH: almacena los 4 bytes de datos restantes.

El funcionamiento del controlador CAN es un tanto diferente al del ESP32, porque en este caso funciona todo mediante registros. Cada Mailbox se configura independientemente de los demás a través de los registros de control. Vamos a proponer un breve ejemplo con ayuda de la Figura 2.10 para explicarlo:

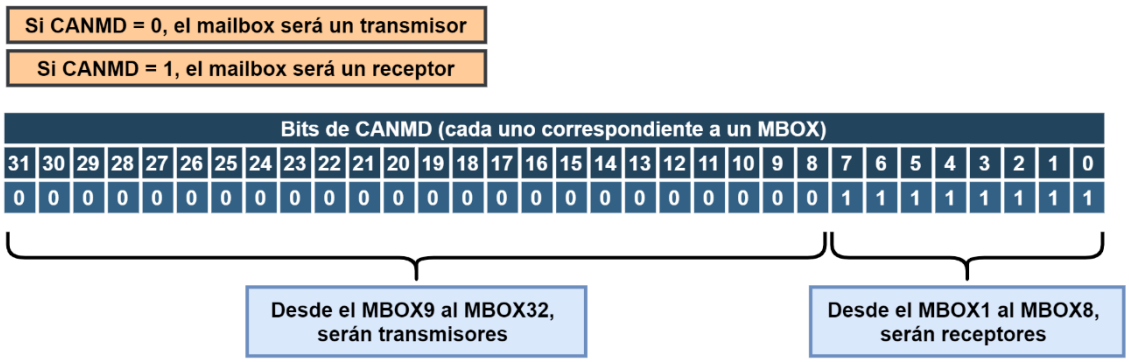

Figura 2.10. Ejemplo de configuración de los registros.

El registro de control CANMD configura los Mailboxes para transmitir o recibir y tiene un tamaño de 32 bits. Cada bit del registro hace referencia a un Mailbox diferente. Según el valor del bit, varía la configuración del Mailbox. En este caso si el bit está a 0 será un

transmisor y a 1 un receptor. Así es posible, por ejemplo, seleccionar que un Mailbox se encargue de transmitir y todos los demás de recibir mensajes.

Texas Instruments también indica que los registros de control y estatus deben modificarse mediante un registro auxiliar que almacene temporalmente la configuración, en lugar de modificar su valor directamente. En la Figura 2.11 podemos ver un ejemplo de ello: primero se asigna la configuración anterior del registro CANME en este registro auxiliar, luego se modifica el registro auxiliar y se vuelve a asignar en el registro original, que es donde surten efecto los cambios.

```
447   /* Enable Mailboxes */
448
449       ECanaShadow.CANME.all = ECanaRegs.CANME.all;
450       ECanaShadow.CANME.all = 0xFFFFFFFF;
451       ECanaRegs.CANME.all = ECanaShadow.CANME.all;
```

Figura 2.11. Modificación de registros en el TMS320

Una vez estos registros de control están configurados, los MBOXs pueden ponerse en funcionamiento y almacenarán los parámetros de la comunicación como el ID, los datos, el número de bytes enviados, etc.

2.2.2 SOFTWARE

Code Composer Studio

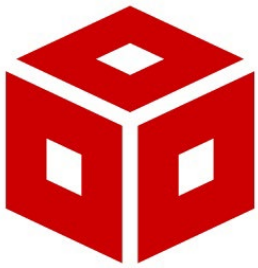

Figura 2.12. Logo CCStudio

Code Composer Studio es un entorno de desarrollo integrado (IDE) para hardware diseñado por Texas Instruments, principalmente microcontroladores y procesadores. Usaremos Code Composer Studio para programar el microcontrolador TMS320F28069, mediante lenguaje C.

PlatformIO

Figura 2.13. Logo PlatformIO

PlatformIO es un entorno de desarrollo integrado (IDE) diseñado especialmente para el internet de las cosas (IoT) y sistemas embebidos. Se trata de una extensión de editores de código como Codeblocks, Eclipse o, en este caso, Visual Studio Code. Ofrece una interfaz muy intuitiva, especialmente para la gestión y configuración de proyectos y nos permite programar en Arduino Framework. Usaremos PlatformIO para programar el ESP32.

Arduino (Plataforma)

Figura 2.14. Logo Arduino

Arduino es una plataforma de código abierto enfocada al desarrollo y la programación simple de hardware y software electrónico. Cuenta con una comunidad enorme y multitud de ejemplos y librerías. El lenguaje de programación que usa la plataforma de Arduino está basado en los lenguajes de programación C y C++, con algunas modificaciones extra para facilitar el manejo con microcontroladores. Aunque no lo usemos en el proyecto, Arduino tiene su propio entorno de desarrollo integrado y también fabrican placas de desarrollo electrónico. Gracias a que es de código abierto, otros fabricantes pueden adaptar sus placas electrónicas para ser compatibles con Arduino, como es el caso del microcontrolador ESP32.

2.2.3 ANALIZADOR CAN BUS

Para analizar la correcta comunicación entre placas se han usado dos accesorios casi imprescindibles a la hora de trabajar con una red CAN: un osciloscopio y un nodo analizador.

2.2.3.1 PC OSCILLOSCOPE

Figura 2.15. Picoscope 2204A

En este caso hemos usado el Picoscope 2204A de la Figura 2.15. Este dispositivo nos permite usar nuestro ordenador como osciloscopio, gracias al software gratuito de Picoscope 6. Nos permite observar las ondas y señales de dos canales, y además es capaz de decodificar los mensajes enviados a través de protocolos seriales como I2C, UART o CAN.

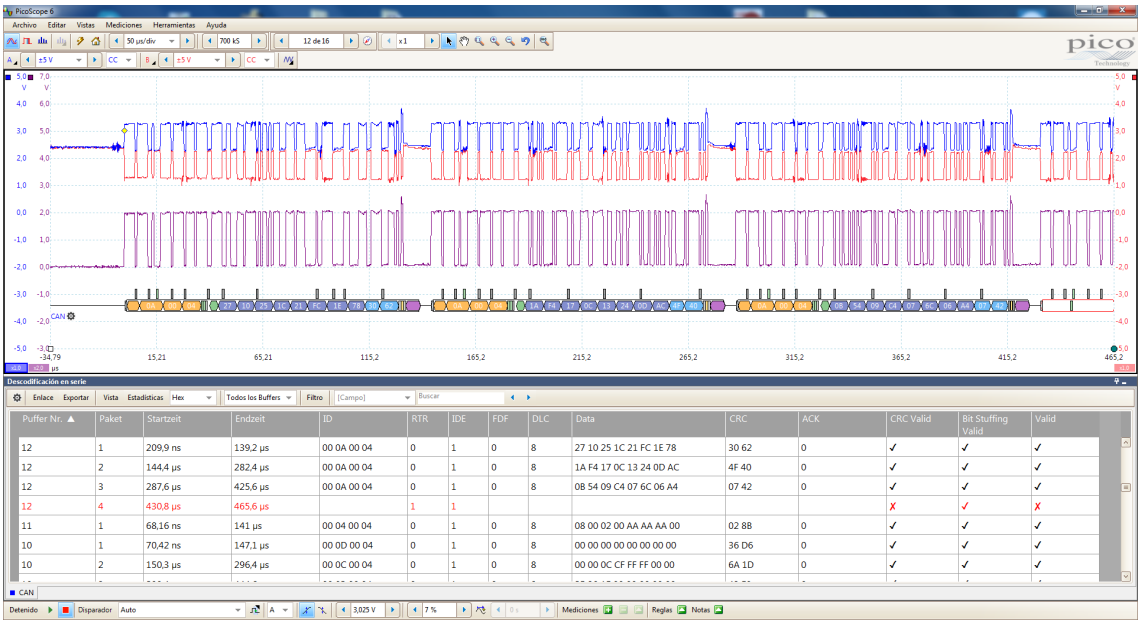

Figura 2.16. Aplicación Picoscope 6.

En la Figura 2.16 podemos observar la aplicación Picoscope 6. En el panel superior podemos configurar ajustes típicos de un osciloscopio como sondas, canales, escalas, muestreo, etc. En el panel inferior se encuentran las configuraciones del “Trigger” o disparo. En la ventana principal se encuentra el gráfico, similar a un osciloscopio, y en este caso hemos añadido también un decodificador de la señal que nos muestra los datos decodificados en una tabla justo debajo.

Este “osciloscopio” en el ordenador es bastante fácil, completo e intuitivo de usar. Tiene algunas limitaciones, como el muestreo de señales de alta frecuencia, pero para nuestro uso no nos hace falta más.

2.2.3.2 ANALIZADOR PCAN

Figura 2.17. Adaptador PCAN USB

El dispositivo PCAN USB de PEAK-System Technik GmbH es un adaptador de CAN bus al PC. Este accesorio convierte, por así decirlo, al ordenador en un “nodo CAN". El

dispositivo, mostrado en la Figura 2.17, requiere del software gratuito PCAN View para monitorear la red CAN, y enviar o recibir mensajes en la red. Esto es útil sobre todo para probar el funcionamiento de un microcontrolador aislado en CAN bus, es decir, para experimentar con él antes de incorporarlo a una red real. También se puede modificar el envío de mensajes de forma muy sencilla.

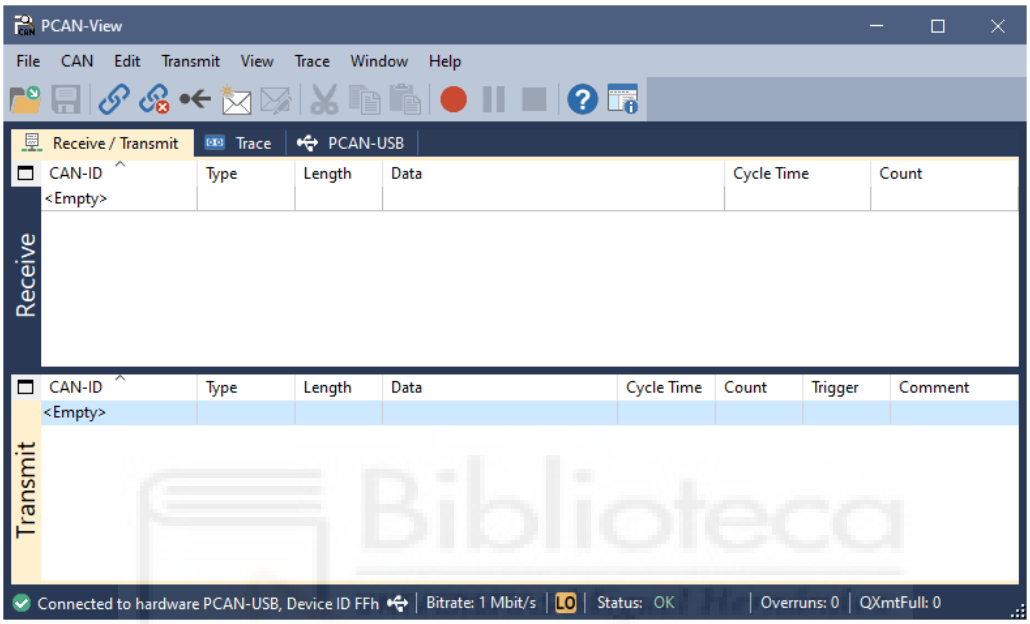

Figura 2.18. Aplicación PCAN-View.

En la Figura 2.18 podemos observar la aplicación PCAN-View, que nos muestra los datos del analizador PCAN-USB. La aplicación es bastante simple. En el panel superior podemos configurar parámetros de la comunicación, como el Baud rate, filtro de aceptación o modo de solo escucha. También podemos crear mensajes para transmitir y grabar la comunicación del bus CAN. En la ventana principal se encuentran los mensajes recibidos y transmitidos. Los mensajes transmitidos son modificables. Además de esta ventana, tenemos una segunda llamada “Trace” para mostrar la grabación del bus: se muestra el tiempo desde el comienzo de la grabación y los datos de los mensajes que han pasado a través del bus, ordenados cronológicamente. La última ventana es para configurar el nombre del puerto USB.

2.2.4 PLACA CONSOLA/INTERFAZ DE USUARIO. ESP32

Esta placa tiene el objetivo de conectar al usuario con el dispositivo USG758 de forma más portátil, mediante una interfaz robusta y sencilla. A través de ella se puede monitorear el estado del USG758 e interactuar con él a varios metros de separación. Esta placa de circuito impreso, mostrada en la Figura 2.19, ha sido diseñada por mi tutor Martin Wassermann. La alimentación es a 24V, aunque hay conversores a 5V y 3.3V para que los componentes electrónicos funcionen correctamente.

La placa se ha diseñado con las siguientes funcionalidades incluidas:

- Microcontrolador ESP32
- Transductor CAN ISO1050
- Convertidor USB-C a RS232
- Pantalla LCD
- Reloj en tiempo real externo
- 6 LEDs programables y 2 pulsadores, ampliable a 8 LEDs y 4 pulsadores
- Adaptador de tarjeta microSD
- 5 botones de navegación
- 2 interruptores giratorios o encoders de 16 posiciones

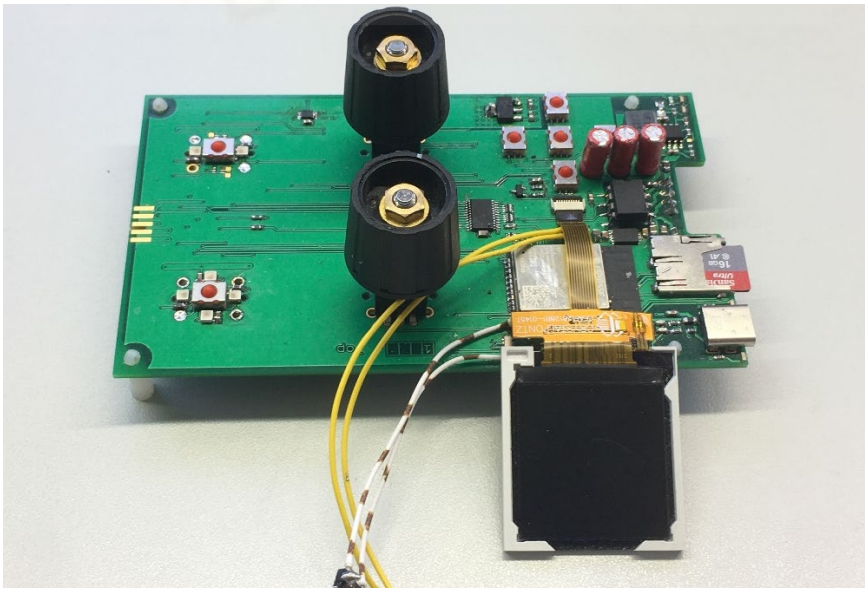

Figura 2.19. Placa electrónica de la consola / interfaz de usuario

Este dispositivo realizará las siguientes acciones a través de la red CAN:

ENVIAR	RECIBIR
Órdenes de ejecución	Almacenamiento de la EEPROM
Modificación de parámetros	Datos de sensores
	Historial de eventos o acciones

2.2.5 DISPOSITIVO USG 758. TMS320F28069.

El dispositivo USG758 es un instrumento electrónico diseñado para controlar el campo magnético de diferentes tipos de imanes. Su placa electrónica dirigida por el TMS320F28069 se encarga principalmente del control de los imanes, aunque también registra y procesa medidas provenientes de sensores de voltaje, corriente y temperatura. Contiene una placa frontal con conectores para control mediante autómatas, LEDs informativos y una pequeña interfaz de usuario. Contiene la electrónica de potencia necesaria para realizar estas acciones, además de una memoria EEPROM (ROM programable y borrable eléctricamente) para guardar la configuración, programas modificables por el usuario y los mensajes de advertencia y error. Contiene un transductor CAN ISO1050 al igual que la placa de la consola.

Este dispositivo realizará las siguientes acciones a través de la red CAN:

ENVIAR	RECIBIR
Almacenamiento de la memoria EEPROM	Órdenes de ejecución
Datos de sensores	Modificación de parámetros
Historial de eventos o acciones	

2.3 CONEXIÓN FÍSICA

En la Figura 2.20 se observa el diagrama de conexión que se ha empleado en el proyecto. El dispositivo USG758 se alimenta de la red eléctrica, y suministra a la consola alimentación a 24V.

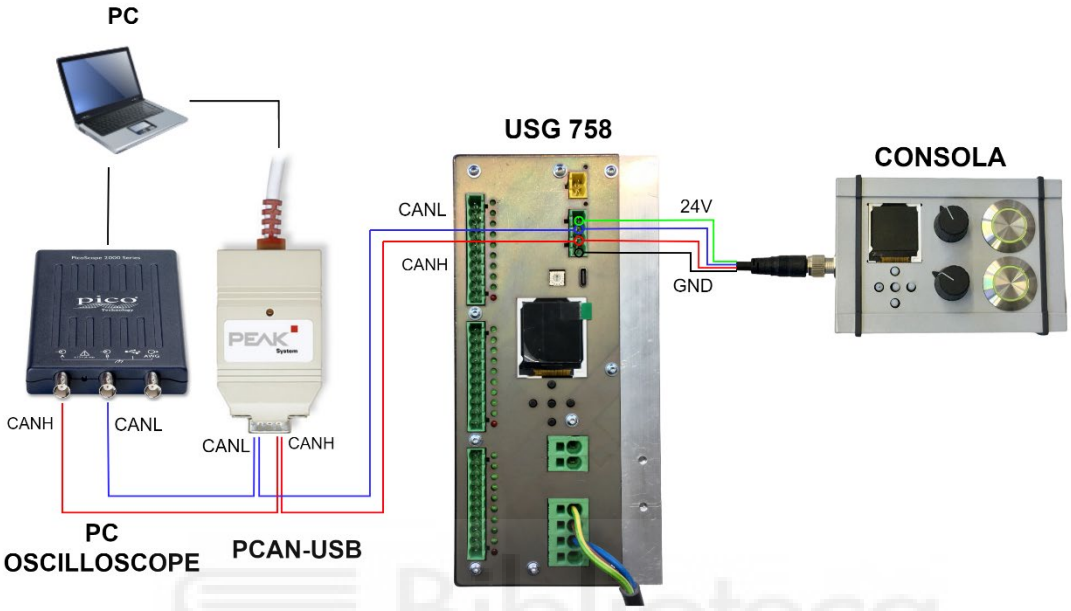

Figura 2.20. Esquema de conexión de la red CAN

La comunicación tiene lugar entre el dispositivo USG758 y la consola, conectados como en la Figura 2.20. El Osciloscopio y el analizador PCAN-USB son prescindibles en la red y se usan como aparato de medida. Esta red es ampliable a más dispositivos siempre que se tenga en cuenta las resistencias de terminación, que deben ser 2 de 120Ω cada una, dispuestas en los extremos, como se explicaba en las características de la capa física en el apartado 2.1. Se ha previsto que haya un máximo de 16 dispositivos USG758 en la red.

Figura 2.21. Esquema del proyecto

En la Figura 2.21 se muestra el esquema de nuestro proyecto. El usuario podrá controlar desde la consola el dispositivo USG758, que a su vez controla el imán. El USG758 enviará de vuelta a la consola los ajustes actuales y medidas de los sensores, que el usuario podrá observar.

2.4 BIBLIOTECAS Y RECURSOS UTILIZADOS

2.4.1 BIBLIOTECAS

Para la implementación de la comunicación CAN en el ESP32, se ha utilizado la biblioteca ESP32-TWAI-CAN desarrollada por Maciej Sorek Loboda, disponible en Github [11]. Esta biblioteca disponible para Arduino implementa el driver TWAI ofrecido por Espressif desde su Guía de programación para su plataforma ESP-IDF [12]. El driver de Espressif ofrece funciones de alto nivel para el usuario, pero internamente usa funciones de bajo nivel (Hardware Abstraction Layer y Low Layer) para acceder al hardware del chip. Debido a su complejidad, no se profundizará tanto en las acciones específicas que se realizan en el código de bajo nivel, sino que se mostrará el funcionamiento descrito en el Manual Técnico de Referencia del ESP32 [13] que debería ser equivalente, en los diagramas de código. Las funciones que se han usado en el proyecto son:

- .begin() : Carga la configuración y habilita al ESP32 para la comunicación CAN.
- .convertSpeed() : Para configurar del Baud rate.
- .readFrame() : recibir un mensaje del búffer.
- .writeFrame() : enviar un mensaje.

Para los microcontroladores C2000, Texas Instruments ofrece un repositorio de software llamado ControlSUITE muy útil para aprender a programar las placas de desarrollo. Este repositorio junto con el informe de aplicación “Programming Examples for the TMS320x28xx eCAN” de Hareesh Janakiraman [14] ha sido utilizado como apoyo para implementar la comunicación CAN en el TMS320F28069.

2.4.2 CREACIÓN Y CONFIGURACIÓN DEL PROYECTO

Ahora que hemos mostrado tanto las aplicaciones como las bibliotecas que utilizaremos en cada microcontrolador, podemos mostrar cómo crear un nuevo proyecto en el que programar todo.

2.4.2.1 PLATFORMIO (ESP32)

Para el ESP32 vamos a necesitar instalado VS Code, y además añadir la extensión PlatformIO. En la bibliografía se encuentra la documentación de la página oficial de

Platformio [15], donde se explica cómo crear un proyecto y otros tutoriales detalladamente.

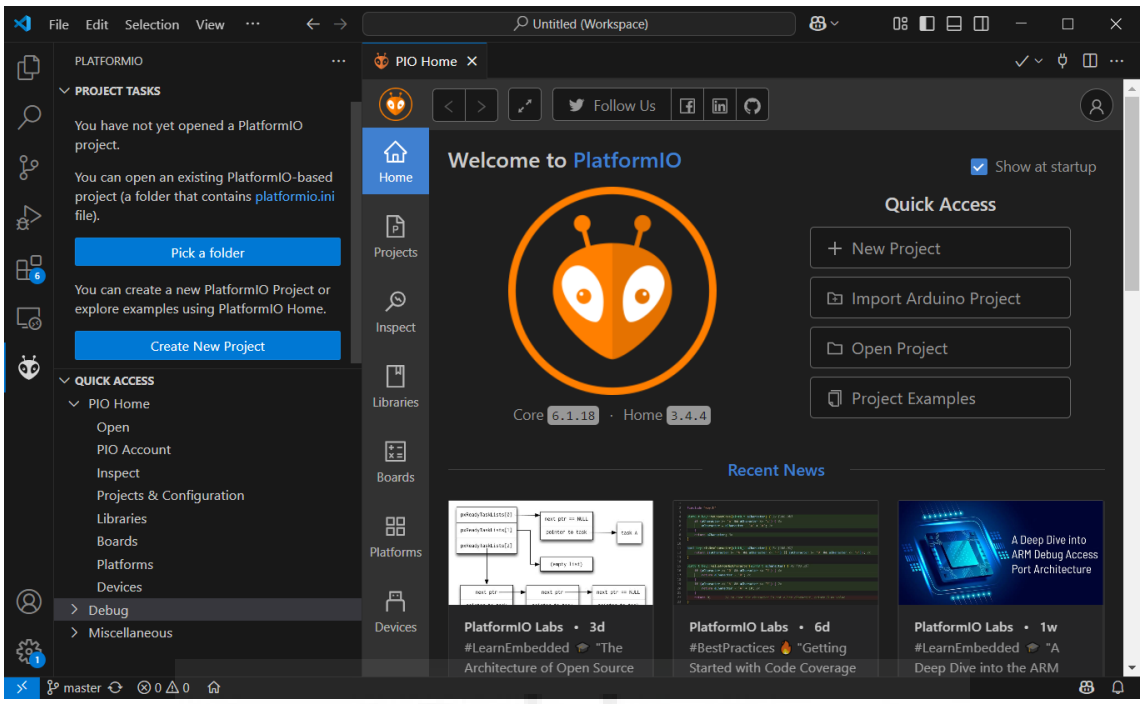

Figura 2.22. PlatformIO - Página principal

En la Figura 2.22 se muestra VS Code con la extensión de Platformio instalada y con la página principal abierta: PIO Home. Desde la página principal podemos crear, importar y abrir nuestros proyectos o ejemplos. También se puede cambiar la configuración de librerías, dispositivos, plataformas, etc.

Para crear un nuevo proyecto, hacemos clic en “New Project” y nos mostrará una ventana igual que en la Figura 2.23.

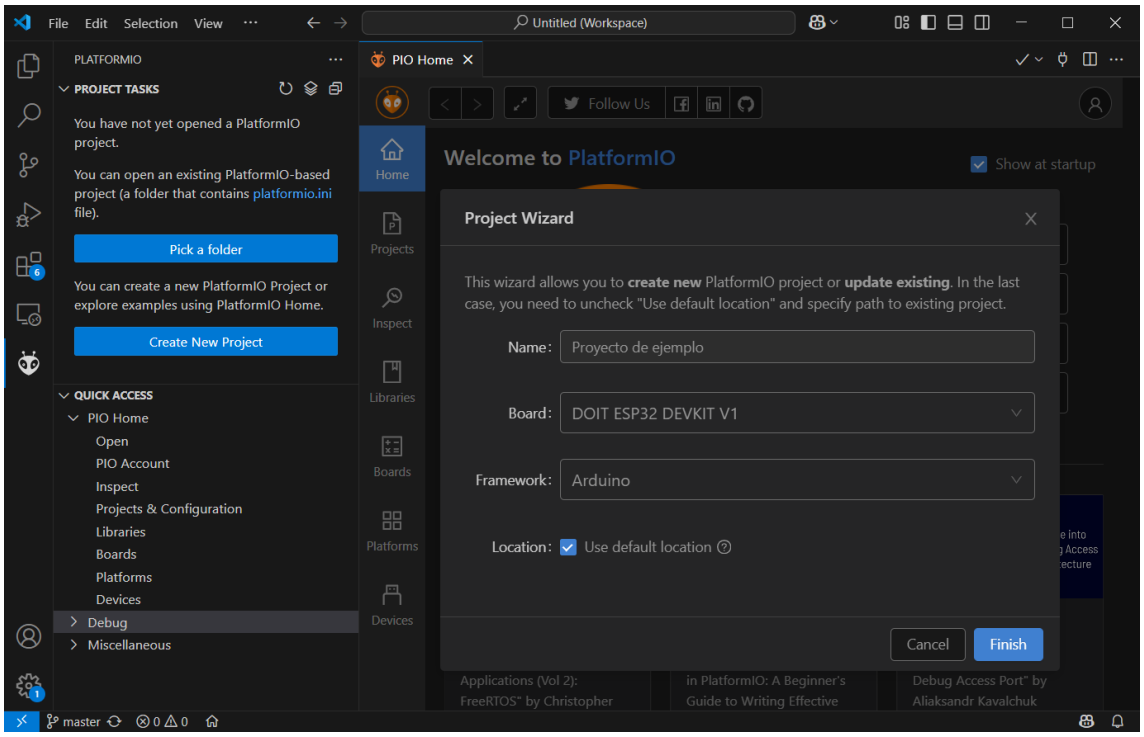

Figura 2.23. PlatformIO. Creación de un nuevo proyecto.

Ahora daremos un nombre a nuestro proyecto, elegiremos la placa, en nuestro caso, “DOIT ESP32 DEVKIT V1”, y luego el Framework, que para el ESP32 puede ser ESP-IDF o Arduino. A continuación, presionamos sobre “Finish” y se nos creará un proyecto nuevo en el “Explorador de documentos”, ubicado en la primera posición en la barra lateral izquierda.

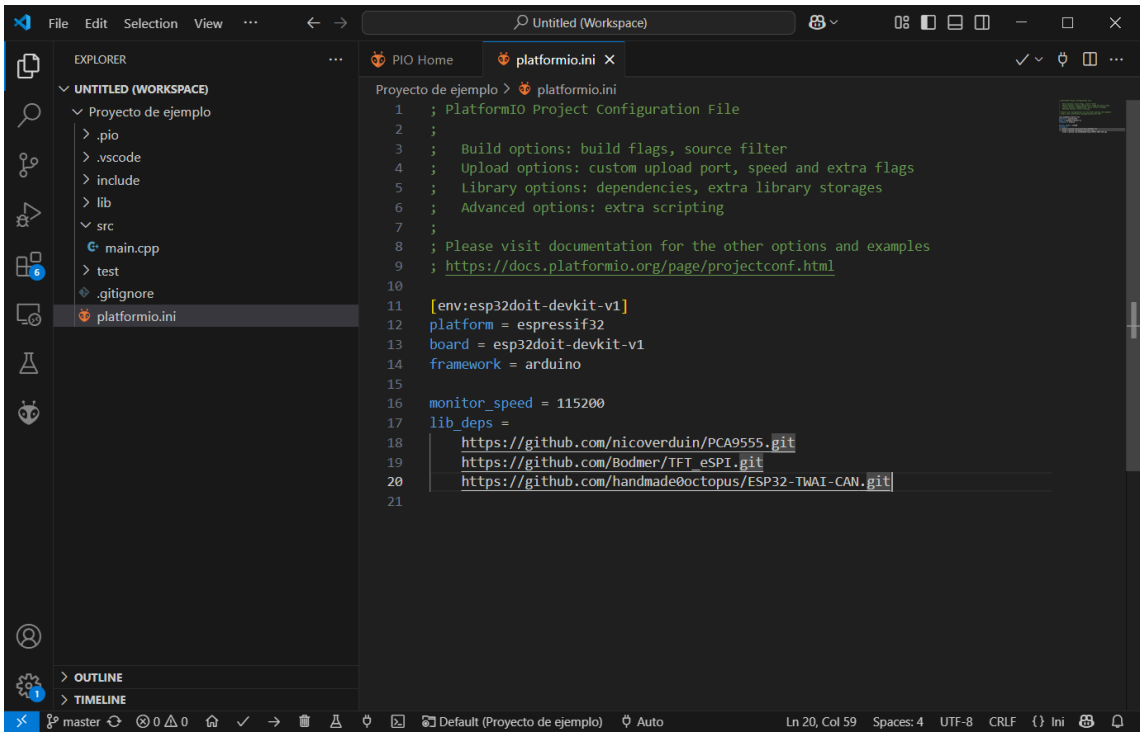

Figura 2.24. PlatformIO - Configuración del proyecto

Si abrimos el proyecto, nos aparecerá lo mismo que en la Figura 2.24 y vemos que se han creado varios directorios:

- ".pio" y ".vscode" almacenan archivos temporales, archivos del compilador o las bibliotecas importadas de internet.
- "include" y "lib" son para incluir headers(.h) y librerías, respectivamente, pero por ahora están vacíos.
- En "src" se ha creado un archivo de código .cpp (c++) con una función main() predeterminada. Aquí escribiremos nuestro código.
- ".gitignore" y "test" no son de nuestro interés.
- Y por último, el archivo "platformio.ini", que almacena la configuración del microcontrolador para nuestro proyecto, nombrado como "esp32doit-devkit-v1". Los parámetros "platform", "board" y "framework" se crean automáticamente con el proyecto. Hemos añadido "monitor_speed = 115200" para poder ver a través del monitor serial del ordenador el resultado de la funcion printf(), útil en ocasiones para observar el flujo de ejecución del programa. Luego hemos añadido "lib_deps" y a continuación el enlace a las librerías de Github que vamos a utilizar. Esto nos almacenará localmente las librerías en el directorio ".pio".

Ahora que tenemos el proyecto creado, podemos empezar a escribir nuestro código.

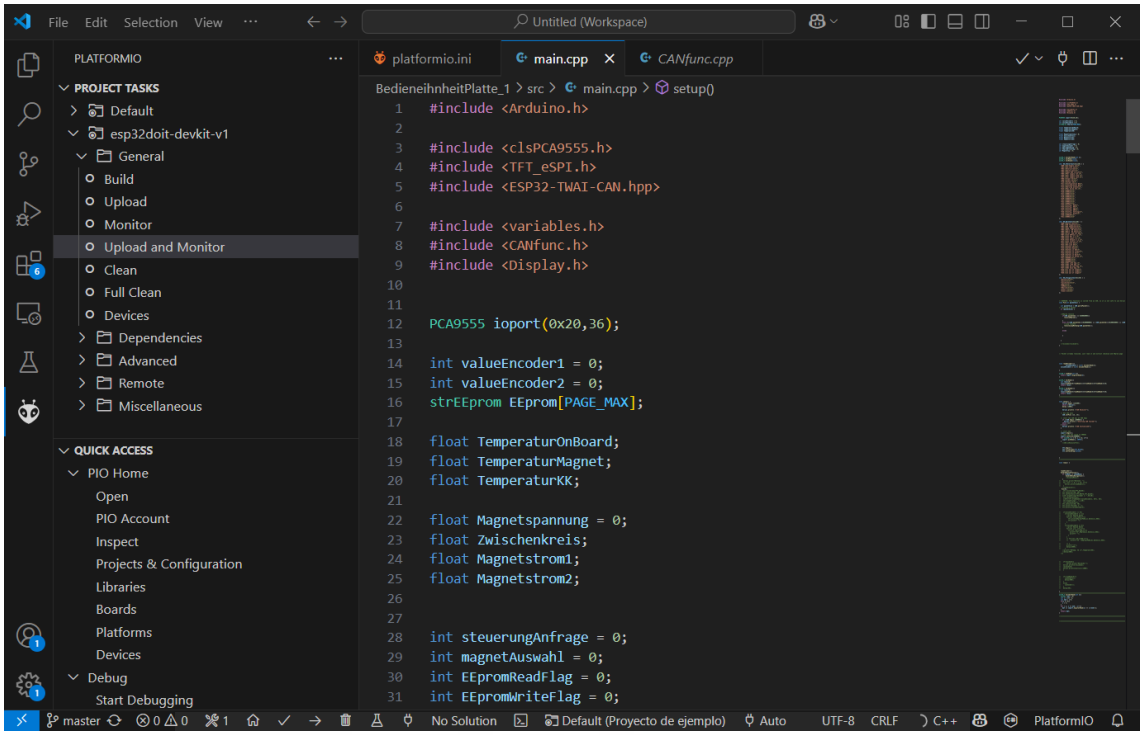

Figura 2.25. PlatformIO - Cargar proyecto en el ESP32

Cuando hayamos escrito nuestro código, podemos volver a la extensión de Platformio (icono en la barra lateral izquierda) y nos aparecerá un icono de una carpeta con la configuración específica del proyecto “esp32doit-devkit-v1”. Desde esta carpeta vamos a ejecutar las acciones:

Build: si nuestro código está correcto, lo compila. Si no, nos muestra el origen del error.

Monitor: abre un monitor serial con el microcontrolador, es decir, nos muestra por consola los datos transmitidos a través del puerto serie. Este tipo de datos se envían cuando se ejecuta algún printf() o Serial.print() en el programa.

Upload and Monitor: compila el código si no lo estaba, y transfiere el programa al microcontrolador. El ESP32 debe estar conectado al ordenador. Una vez cargado el programa en el microcontrolador, se abre el Monitor serial.

Tras haber cargado el programa, este se ejecutará constantemente incluso tras desconectar el ESP32 del ordenador. Si hiciera falta resetear el programa, los microcontroladores suelen tener un mecanismo o puerto para ello, o algún botón si se trata de una placa de desarrollo.

2.4.2.2 CODE COMPOSER STUDIO (TMS320F28069)

En el caso del TMS320F28069, crear un proyecto es bastante más complejo, ya que hay que añadir las cabeceras(.h) predeterminadas que ofrece Texas Instruments desde ControlSUITE, configurarlas posteriormente en CCS y luego escribir nuestro código. En ControlSUITE también hay un documento que explica paso a paso cómo crear un proyecto. Así que, debido a la complejidad de crear un proyecto para el TMS320, si alguien tiene más interés puede buscar este documento o tutorial en la bibliografía [16] o en ControlSUITE, como se muestra en la Figura 2.26:

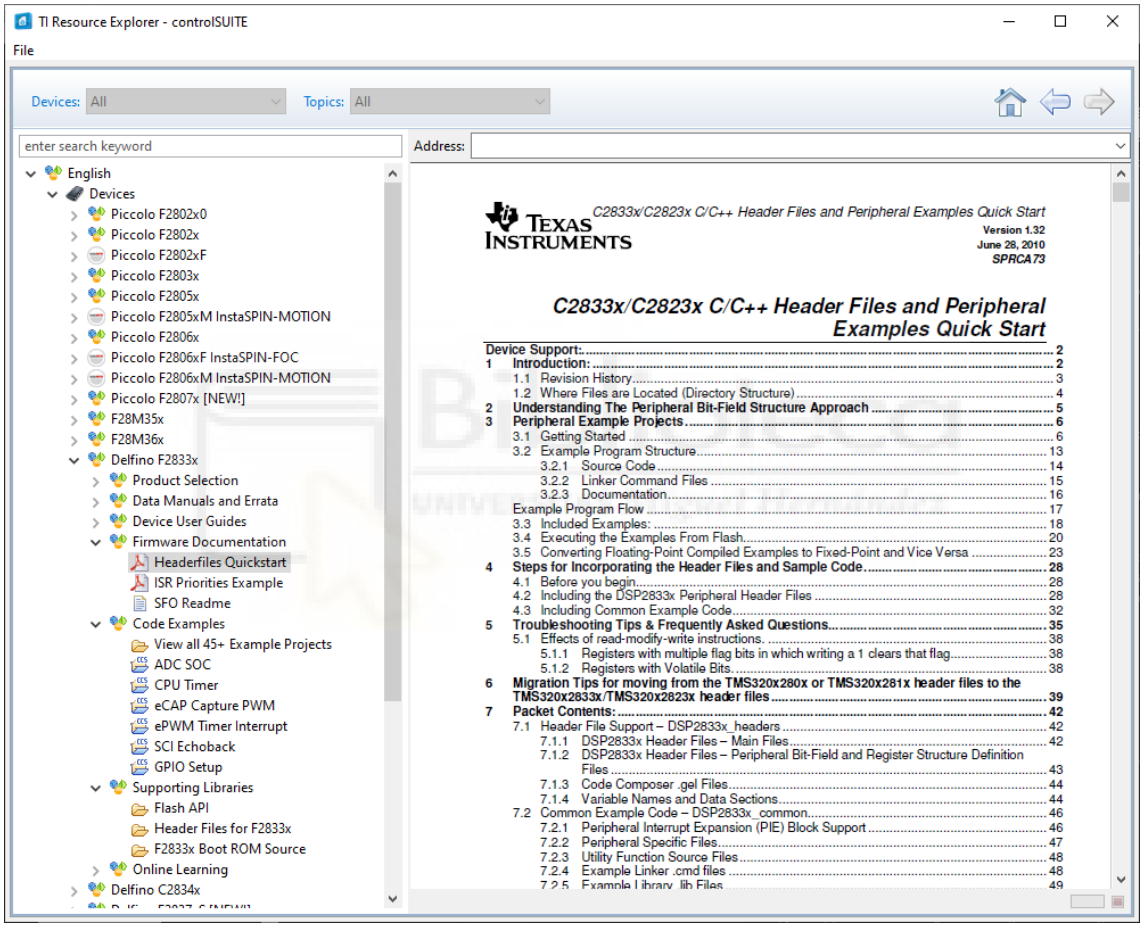

Figura 2.26. ControlSUITE – Ubicación de las instrucciones para crear un proyecto

Otra opción es importar un proyecto ya creado de los "Ejemplos de programación del módulo eCAN para el TMS320x28xx" [14] y a partir de ahí añadir nuestro código. Una vez creado el proyecto y nuestro código, presionando clic derecho sobre el proyecto, debemos ir a "Propiedades". Ahí podemos cambiar el modelo del microcontrolador y el Debugger al nuestro, tal y como se muestra en la Figura 2.27

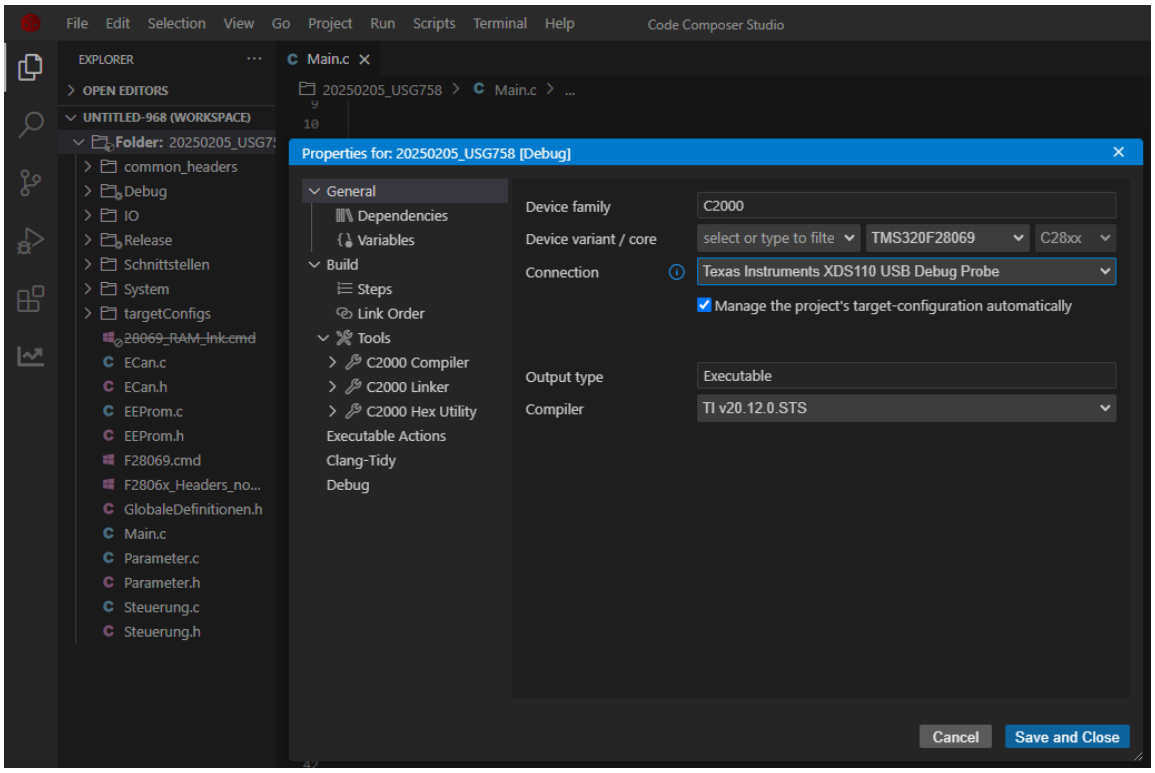

Figura 2.27. CCS – Propiedades del proyecto

Una vez ajustadas las propiedades, el proyecto estará listo para cargarse en el microcontrolador.

2.5 DESARROLLO SOFTWARE

2.5.1 CONFIGURACIÓN DE BIT TIMING

Antes de la ejecución del programa principal sucede la inicialización del módulo CAN y ahí se selecciona el Baud Rate empleado en la comunicación. En nuestro caso, la comunicación tendrá un Baud Rate de 1Mbps y nuestro reloj del controlador CAN funciona a 40 MHz. Para que el nodo trabaje al Baud Rate deseado, se requiere un control preciso del Bit timing, es decir, controlar la duración y retardo específicos de cada bit.

El TMS320F28069 tiene el registro CANBTC (CAN Bit-Timing Configuration) donde se almacena la configuración del Bit Time. Texas Instruments ofrece una hoja de cálculo [17] para la configuración del Bit Time en la serie de microcontroladores C2000 muy útil y sencilla. Es necesario introducir el Baud Rate deseado, la longitud del bus, frecuencia de funcionamiento del reloj CAN y retardo de propagación, aunque este último parámetro lo dejaremos por defecto.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		**CAN Bit Timing Calculator**											
2		Copyright 2016 by Texas Instruments Inc.											
3		This tool is intended for use as a calculation aid. You are ultimately responsible for the bit timing settings on your CAN network.											
4													
5		**Input bus parameters here**											
6		**Bit rate (kilobits/sec)**	**1000**										
7		**Bus length (m)**	**5**		Assumed propagation speed is 5 ns/m								
8		**Extra propagation delay (us)**	**0.1**		From isolation or other sources of physical layer delay								
9		**CAN clock frequency (MHz)**	**40**		SYSCLK/2 for eCAN or the CAN bit clock frequency for D_CAN								
10													
11													
12		**Output**											
13		Bit time (Tq)	10		These settings are chosen to achieve the loosest oscillator tolerance								
14		Bit clock prescaler	4										
15		Propagation segment (Tq)	3										
16		Phase segment 1 (Tq)	3										
17		Phase segment 2 (Tq)	3										
18		Synchronization jump width (Tq)	3										
19		Sampling point (%)	70.0%										
20													
21													
22		**Oscillator tolerance**	**1.18%**		The tolerance value shown includes any error in the nominal bit rate								
23		**D_CAN CAN_BTR register value**	**0x02583**										
24		**eCAN CAN_BTC register value**	**0x03022A**										

Figura 2.28. Calculadora de Bit Timing CAN [17]

Tras introducir los datos, en nuestro caso como en la Figura 2.28, nos devuelve exactamente el valor completo del registro CAN_BTC, utilizado en la inicialización del módulo CAN: 0x03022A.

Por otro lado, el controlador CAN del ESP32 tiene dos registros para almacenar estos parámetros del Bit Time, BTR0 y BTR1. Pero el driver TWAI del ESP32 ofrece unos

valores por defecto del Bit Timing para ciertos Baud Rates, entonces no se ha requerido ajustarlo. Existen calculadoras en internet para el Bit Time del controlador CAN SJA1000, que es supuestamente equivalente al controlador CAN del ESP32.

2.5.2 USO DE IDENTIFICADORES

En vista a controlar varios dispositivos desde una sola consola, surgía la cuestión de cómo se comunicarían entre ellos, y aquí es donde el identificador o ID juega un papel importante. Cabe recalcar que en el bus CAN los dispositivos no tienen identificador propio, sino que los mensajes tienen identificador, es decir, un dispositivo puede enviar mensajes con diferentes IDs. Los dispositivos USG758 tendrán el mismo programa, por tanto, todos los dispositivos usarían los mismos IDs: la consola recibiría órdenes de varios dispositivos sin saber cuál es el emisor exactamente, o se enviaría mensajes a todos los dispositivos a la vez sin control del receptor.

Esto es un gran inconveniente, y para ello se ha pensado en dedicar una parte del identificador al nº de dispositivo. En un frame extendido, tenemos 29 bits configurables en el identificador, frente a 11 bits en un frame normal. Se espera tener máximo 16 dispositivos USG758 conectados a la red, así que con 4 bits (hasta 16 opciones) podría servirnos para saber el nº de dispositivo. Pero de esta forma solo se podría enviar órdenes a un dispositivo al mismo tiempo. Como es interesante poder controlar varios dispositivos simultáneamente desde la consola, se ha pensado en dedicar 16 bits del identificador al nº de dispositivo, un bit para cada dispositivo. Funcionaría de la siguiente forma:

Cada dispositivo USG758 tendrá almacenado en la EEPROM su nº de dispositivo, y en base a este número modificará su identificador. Por ejemplo, si tenemos el dispositivo nº 7, el séptimo bit del identificador de todos los MBOXs del USG758 tendrá el valor 1. Por tanto, el dispositivo 7 siempre transmitirá y recibirá los mensajes con el bit 7 del identificador activo.

	Bits															
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Identificador dispositivo 7	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Figura 2.29. Bits de numeración de dispositivo en el identificador

Pero todavía hay un problema, si la consola quiere enviar datos a dos dispositivos, el 2 y el 7 por ejemplo, su identificador tendrá activo el bit 2 y 7. Pero los MBOXs solo reciben mensajes si el identificador coincide exactamente en cada bit. Solo se recibirán mensajes si el bit 7 está activo y el resto de bits inactivos. Para solucionar esto podemos emplear una máscara de aceptación de bits, haciendo que el mensaje se reciba si tiene el bit 7 activo y no importe el valor del resto de bits.

	Bits															
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Identificador dispositivo 7	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
Máscara de aceptación	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
Mensajes recibidos	X	X	X	X	X	X	X	X	X	1	X	X	X	X	X	X

Figura 2.30. Ejemplo de uso de máscara de aceptación en el identificador

Con esta máscara de aceptación, el dispositivo 7 solo recibirá los mensajes que tengan el bit 7 del identificador activo, y no recibirá mensajes si el bit 7 está inactivo. Por otro lado, la consola podrá enviar un mensaje que reciban los 16 dispositivos si así se desea.

Entonces ahora desde la consola seremos capaces de saber de qué dispositivo provienen los mensajes y guardar los datos en función de esto, además de poder enviar mensajes a varios USG758 simultáneamente.

Aparte de usar una parte del identificador para diferenciar cada dispositivo, se ha pensado en usar otra parte para diferenciar la funcionalidad del mensaje. Esto es similar al funcionamiento del Identificador en `CANopen`, donde se reserva una parte del ID para señalar que función tiene el mensaje, por ejemplo, emergencia, sello de tiempo o configuración. Usaremos 4 bits con esta finalidad, que nos permite hasta 16 opciones. Cada MBOX del USG758 tendrá asignada una función, la cual se muestra en la Tabla 3, como enviar mensajes cíclicos, eventos, el contenido de la memoria EEPROM, o recibir valores. En la Tabla 3 se muestra los identificadores previstos y su función, aunque siempre se pueden añadir más.

	ID (hex)	Función
Consola → USG758	0100 (0x4)	Modificar variables cuando el usuario lo selecciona
USG758 → Consola	1010 (0xA)	Se envía el EEPROM
	1011,1100,1101 (0xB,0xC,0xD)	Se envían mensajes cíclicos con datos de sensores
	1110 (0xE)	Se envían eventos (Fallos, advertencias, instrucciones)

Tabla 3. Identificadores utilizados y su función

Tras esto explicado, se puede mostrar un ejemplo de cómo se asignaría un identificador completo en la Figura 2.31 y la Figura 2.32 y de cómo se asignaría su máscara de aceptación correspondiente en la Figura 2.33:

```
390     ECanaMboxes.MBOX4.MSGID.all = 0x00040004;   //RECEIVE
391
392     ECanaMboxes.MBOX10.MSGID.all = 0x000A0004;  //EEPROM
393
394     ECanaMboxes.MBOX11.MSGID.all = 0x000B0004;  //
395     ECanaMboxes.MBOX12.MSGID.all = 0x000C0004;  //PROZESSDATEN
396     ECanaMboxes.MBOX13.MSGID.all = 0x000D0004;  //
397
398     ECanaMboxes.MBOX14.MSGID.all = 0x000E0004;  //WARNUNG FEHLER
```

Figura 2.31. Asignación del ID en código

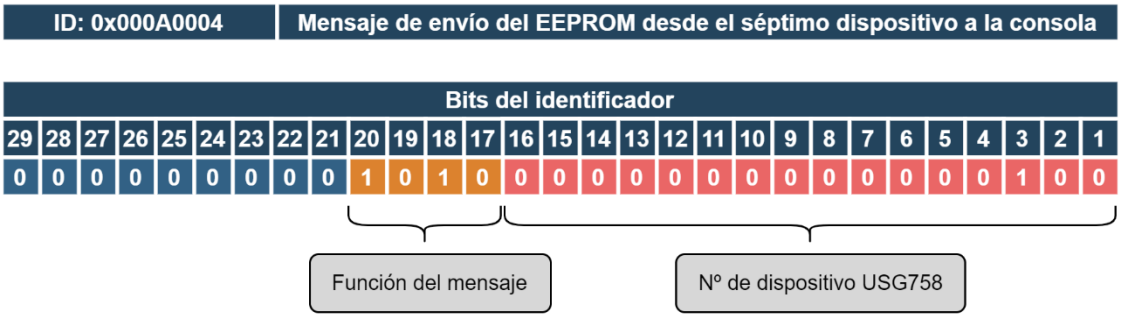

Figura 2.32. Ejemplo de un identificador completo y su estructura

```
423     ECanaLAMRegs.LAM4.all = 0b11111111100001111111111111011;
```

Figura 2.33. Asignación de la máscara de aceptación en código

Los identificadores siempre tendrán 16 bits para el número de dispositivo, 4 bits para la función del mensaje, y quedarán 9 bits libres.

2.5.3 ALMACENAMIENTO DE LA CONSOLA

Desde la consola se ha previsto poder controlar hasta 16 dispositivos diferentes. Para ello, se requiere de un determinado espacio para guardar la configuración de cada dispositivo almacenada en la EEPROM. Se ha creado unas estructuras con esta finalidad y una enumeración para facilitar la legibilidad, las cuales podemos ver en la Figura 2.34:

```
8   typedef enum {
9       PAGE_NOT_FOUND = 0,
10      PAGE_1,
11      PAGE_2,
12      PAGE_3,
13      PAGE_4,
14      PAGE_5,
15      PAGE_6,
16      PAGE_MAX
17  } EEprom_page_t;
18
19  typedef struct {
20      uint8_t Daten[256];
21      uint8_t checksum;
22  } strEEprom;
23
24  typedef struct {
25      uint16_t ID;
26      strEEprom EEprom[PAGE_MAX];
27  } strDevice;
28
```

Figura 2.34. Declaración de las estructuras y enumeración

La estructura `strEEprom` almacena los datos y el checksum de una página de la memoria EEPROM. La estructura `strDevice` almacena la parte del identificador correspondiente al nº de dispositivo y un array o arreglo de páginas de EEPROM equivalente a la memoria EEPROM completa. Las variables de tipo `strDevice` consumen una cantidad considerable de memoria, alrededor de 1,5KB cada una. El ESP32 está lejos de ocupar la memoria disponible con este programa, pero si hubiera que reducirla, la variable `strDevice` se podría tener en cuenta.

2.5.4 DIAGRAMAS DE CÓDIGO

Para facilitar la lectura y comprensión del funcionamiento de los programas, se expondrá su código mediante diagramas de flujo ordenados de más general a más específico, en algunos casos mostrando la interacción entre ambos chips. Así se proporciona una visión clara de la estructura del software, permitiendo identificar en primer lugar los procesos principales y luego profundizar en cada componente específico.

El código expuesto será principalmente sobre la comunicación CAN. La explicación de otras acciones como el control de los imanes o el funcionamiento de los menús se omitirán, ya que salen del interés del proyecto, aunque se ha tenido en cuenta que no se vean afectadas por el nuevo código.

A continuación, en la Figura 2.35, se presenta el flujo general del programa donde se describen los procesos principales. En las posteriores secciones, cada proceso o conjunto de acciones será desglosado en mayor detalle.

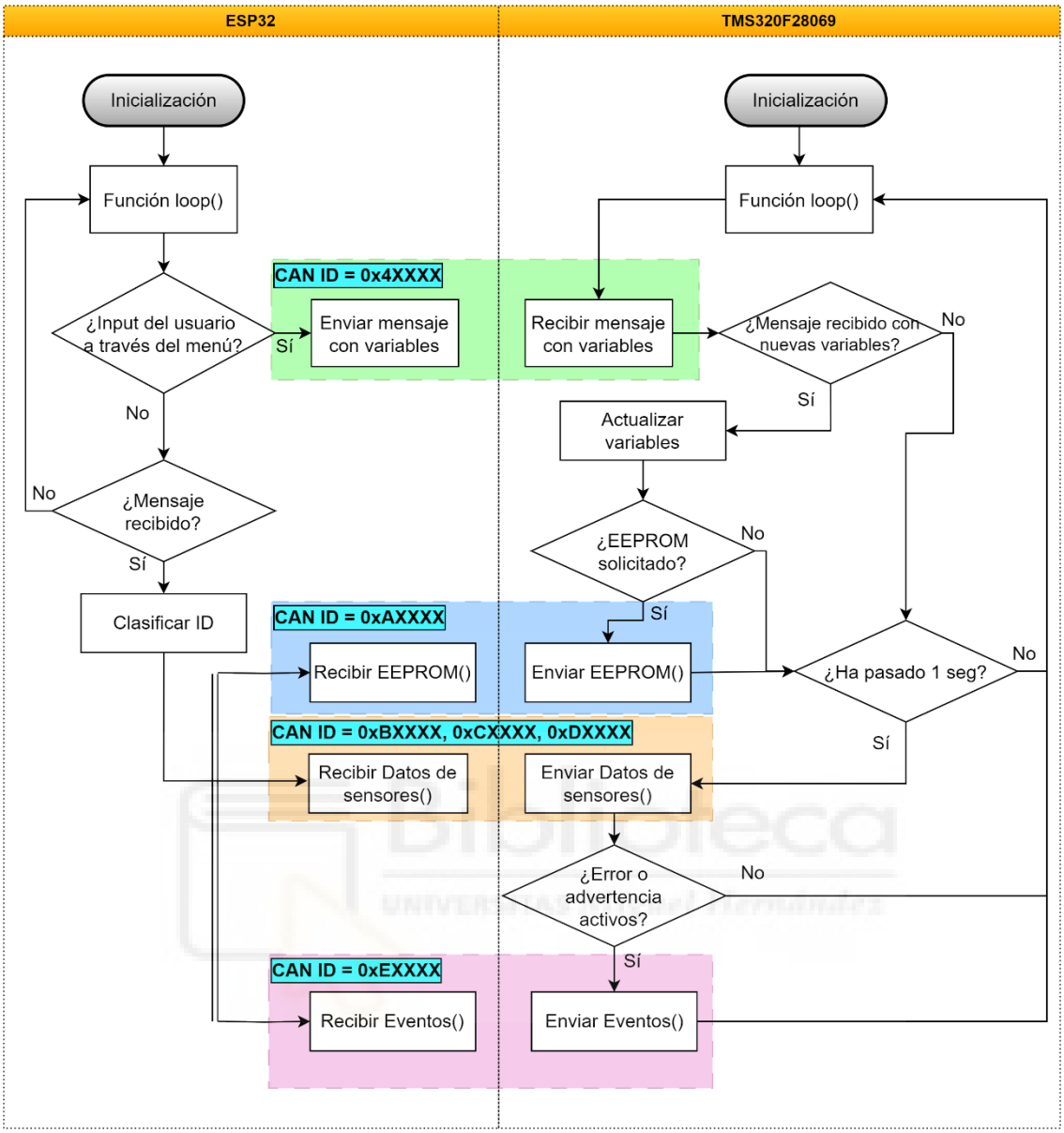

Figura 2.35. Diagrama de flujo del programa principal

Sobre el diagrama principal cabe destacar que la inicialización se realiza una sola vez tras encender o resetear los dispositivos, y en ella se realizan principalmente configuraciones, iniciación de módulos como CAN o vaciado de variables y memoria a cero. La "Función loop()" es en sí la función principal que se ejecuta de forma repetitiva o cíclica y contiene en su interior prácticamente todo el código del programa.

El siguiente diagrama desglosa el proceso de inicialización, mostrando las configuraciones necesarias antes de la ejecución principal.

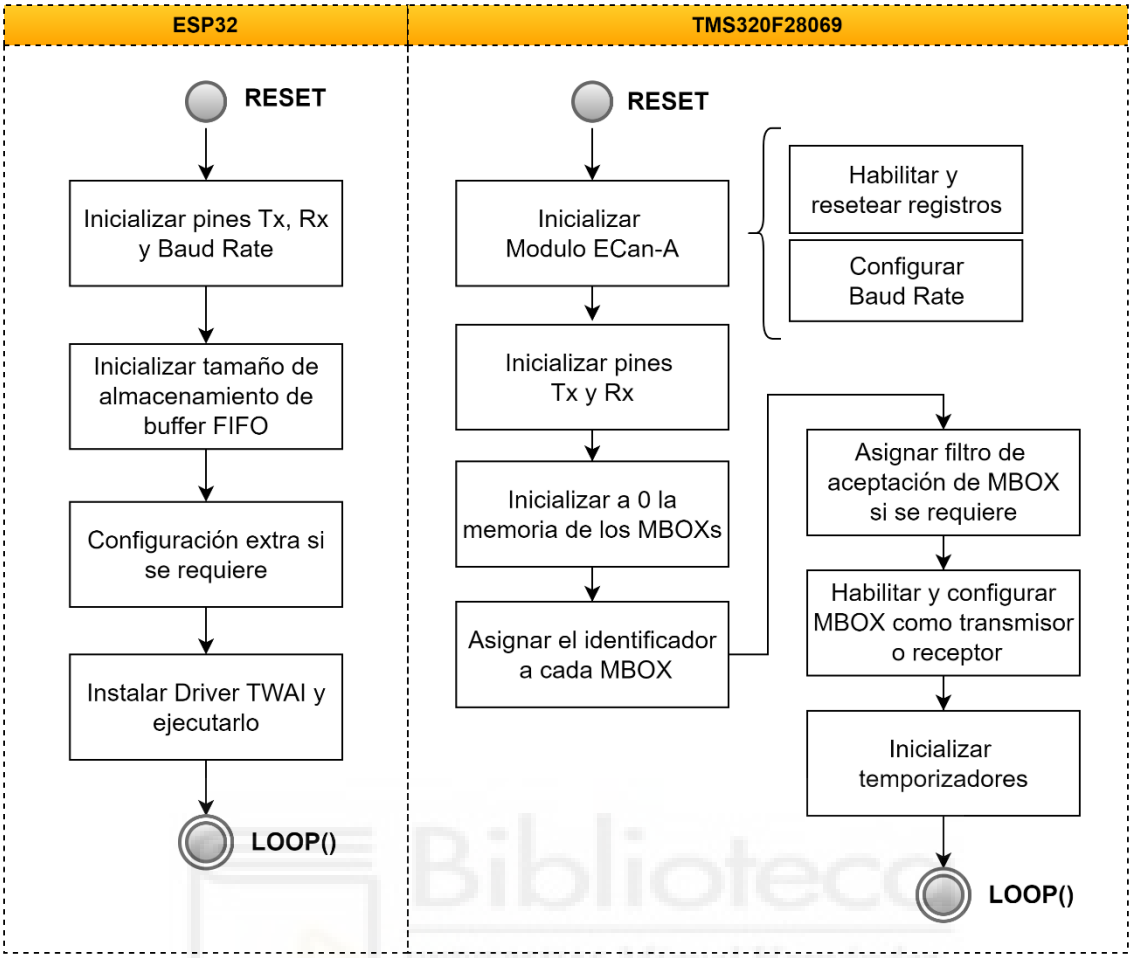

Figura 2.36. Diagrama del proceso de inicialización.

Los pines Tx y Rx hacen referencia a los puertos de comunicación o GPIOs dedicados a transmitir y recibir mensajes, respectivamente. Con "Instalar Driver TWAI" nos referimos a reservar el espacio de almacenamiento necesario para guardar las funciones y variables del módulo TWAI.

Una vez inicializados los microcontroladores, el primer proceso es comprobar si el usuario ha generado una orden. Si así es, se ejecutará el código mostrado a continuación en la Figura 2.37:

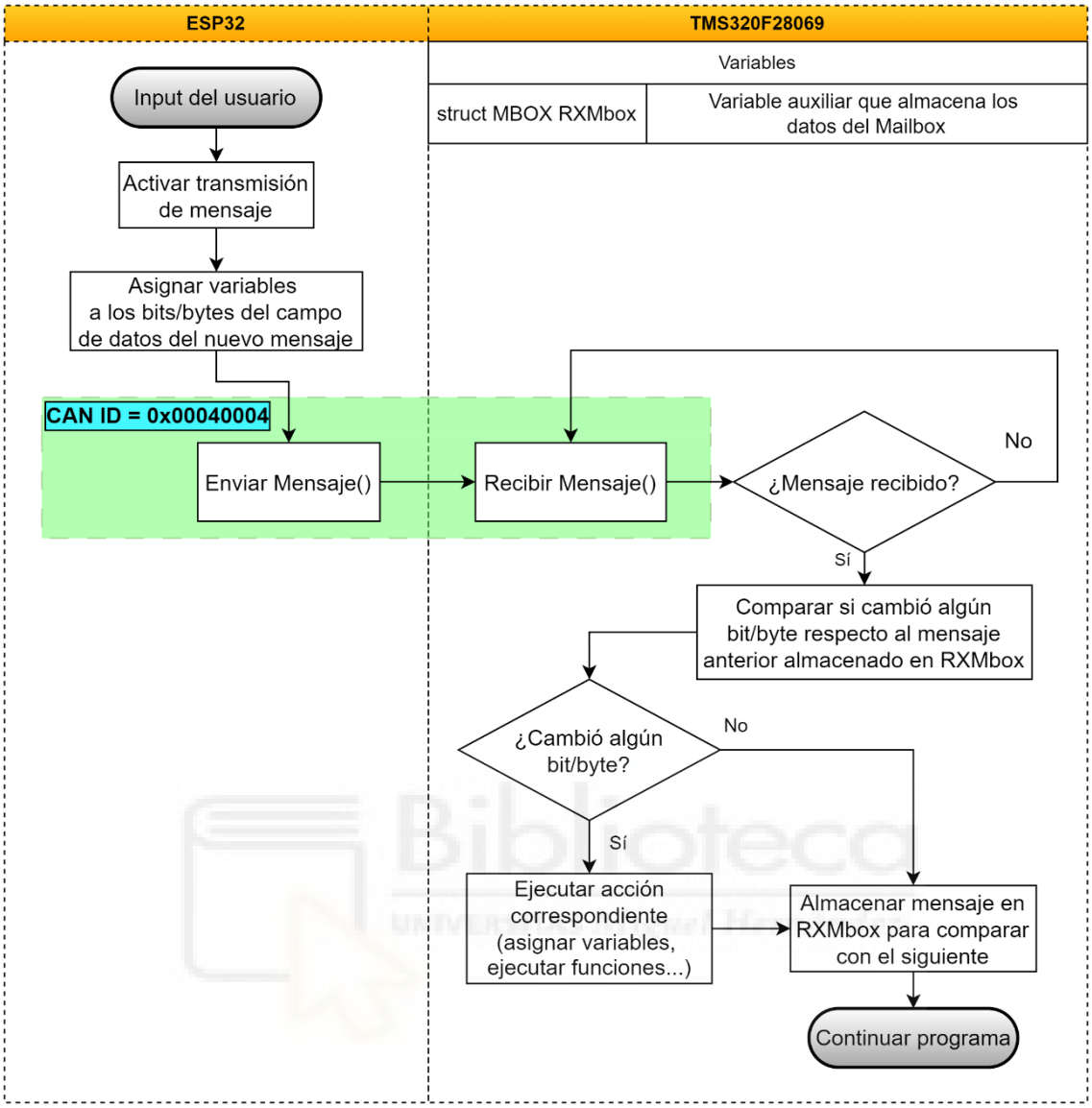

Figura 2.37. Diagrama del proceso de mandar instrucciones

El usuario será capaz de modificar variables desde la consola. Cuando se genere un input, por ejemplo, dar la instrucción de "Magnetizar imán", la variable que almacena la instrucción en curso recibirá el valor correspondiente con el de "Magnetizar imán". Luego en el campo de datos del mensaje CAN, habrá un byte reservado específicamente para esta variable, como se observa en la Figura 2.38. Se enviará el mensaje y el receptor comprueba directamente si cada byte fue modificado. Si el byte correspondiente a la instrucción en curso fue modificado, se ejecutará una acción en base a su valor.

```
33  void canSendMessage(uint32_t Id = 0x00040004) {
34      Serial.print ("Sending packet 2 ... ");
35
36      CanFrame txFrame          = {0};
37      txFrame.identifier        = Id;
38      txFrame.extd              = 1;
39      txFrame.data_length_code  = 8;
40
41      txFrame.data[0]           = steuerungAnfrage;
42      txFrame.data[1]           = magnetAuswahl;
43      txFrame.data[2]           = EEpromReadFlag;
44      txFrame.data[3]           = Magnettyp;
45      txFrame.data[4]           = 0xAA;
46      txFrame.data[5]           = 0xAA;
47      txFrame.data[6]           = 0xAA;
48      txFrame.data[7]           = EEpromWriteFlag;
49
50      if(ESP32Can.writeFrame(txFrame)){
51          Serial.println("done");
52      }
53      else{
54          Serial.println("failed");
55      }
```

Figura 2.38. Fragmento de código – Envío del input del usuario desde el ESP32

Una de las acciones más relevantes que el usuario puede generar como input es solicitar los datos de la memoria EEPROM. Estos datos se transmiten también a través del CAN bus. El siguiente diagrama de la Figura 2.39 muestra las acciones que se ejecutan para la lectura del contenido de la EEPROM y es, por así decirlo, una extensión del anterior diagrama.

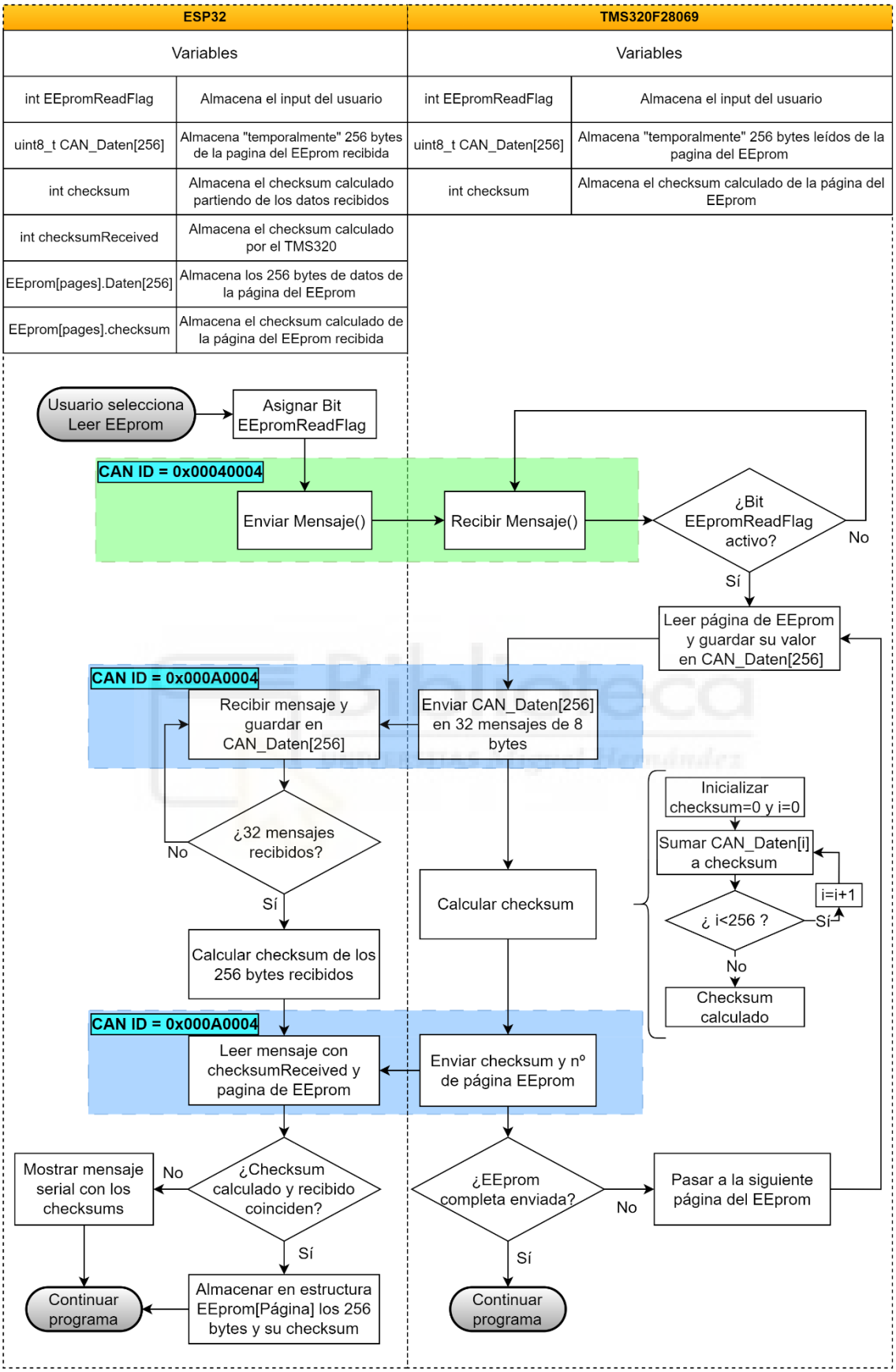

Figura 2.39. Diagrama del proceso de transmisión de la memoria EEPROM

Cuando el usuario solicita la lectura de la memoria EEPROM, el USG758 transfiere a través del CAN bus la memoria EEPROM compuesta de 6 páginas con 256 bytes cada una. Cada página se lee de la EEprom, se almacena en `CAN_Daten[]` temporalmente y se envía. Para enviar cada página se necesitan 32 mensajes por página.

```
213  void sendCANDatenEEprom(int page){
214      int p=0;
215      int i=0;
216      for(i=0; i<32; i++){
217          p=i*8;
218          ECanaMboxes.MBOX10.MDL.byte.BYTE0 = CAN_Daten[p];
219          ECanaMboxes.MBOX10.MDL.byte.BYTE1 = CAN_Daten[p+1];
220          ECanaMboxes.MBOX10.MDL.byte.BYTE2 = CAN_Daten[p+2];
221          ECanaMboxes.MBOX10.MDL.byte.BYTE3 = CAN_Daten[p+3];
222
223          ECanaMboxes.MBOX10.MDH.byte.BYTE4 = CAN_Daten[p+4];
224          ECanaMboxes.MBOX10.MDH.byte.BYTE5 = CAN_Daten[p+5];
225          ECanaMboxes.MBOX10.MDH.byte.BYTE6 = CAN_Daten[p+6];
226          ECanaMboxes.MBOX10.MDH.byte.BYTE7 = CAN_Daten[p+7];
227
228          // Transmit starts
229          ECanaShadow.CANTRS.all = 0;
```

Figura 2.40. Fragmento de código – Envío de la EEprom en el TMS320

Tras enviar cada página se realiza una comprobación o “checksum”, para asegurarse de que se han recibido correctamente los datos y se envía en el mensaje nº33. En total se envían 198 mensajes para la memoria EEPROM completa (33 mensajes * 6 páginas). Si el checksum es correcto, se almacena la página de la EEPROM en la consola.

Si no hay ninguna entrada hecha por el usuario, el programa continúa su ejecución.

Los microcontroladores son capaces de gestionar el tiempo mediante variables de manera sencilla. Para ello, se puede declarar una variable que almacene un valor inicial que represente el tiempo en milisegundos y, mediante un temporizador interno, reducir su valor en una unidad cada milisegundo, permitiendo así medir intervalos de tiempo con precisión.

Se ha programado que se envíen datos cíclicamente cada segundo. Por tanto, la próxima instrucción que se realiza es comprobar si ha pasado un segundo desde la última vez que se enviaron estos datos.

Hay dos tipos de datos enviados periódicamente:

- Datos de sensores
- Eventos tales como errores, advertencias o algunas instrucciones.

En el diagrama de la Figura 2.41 a continuación se muestra más detalladamente cómo se transmiten estos datos:

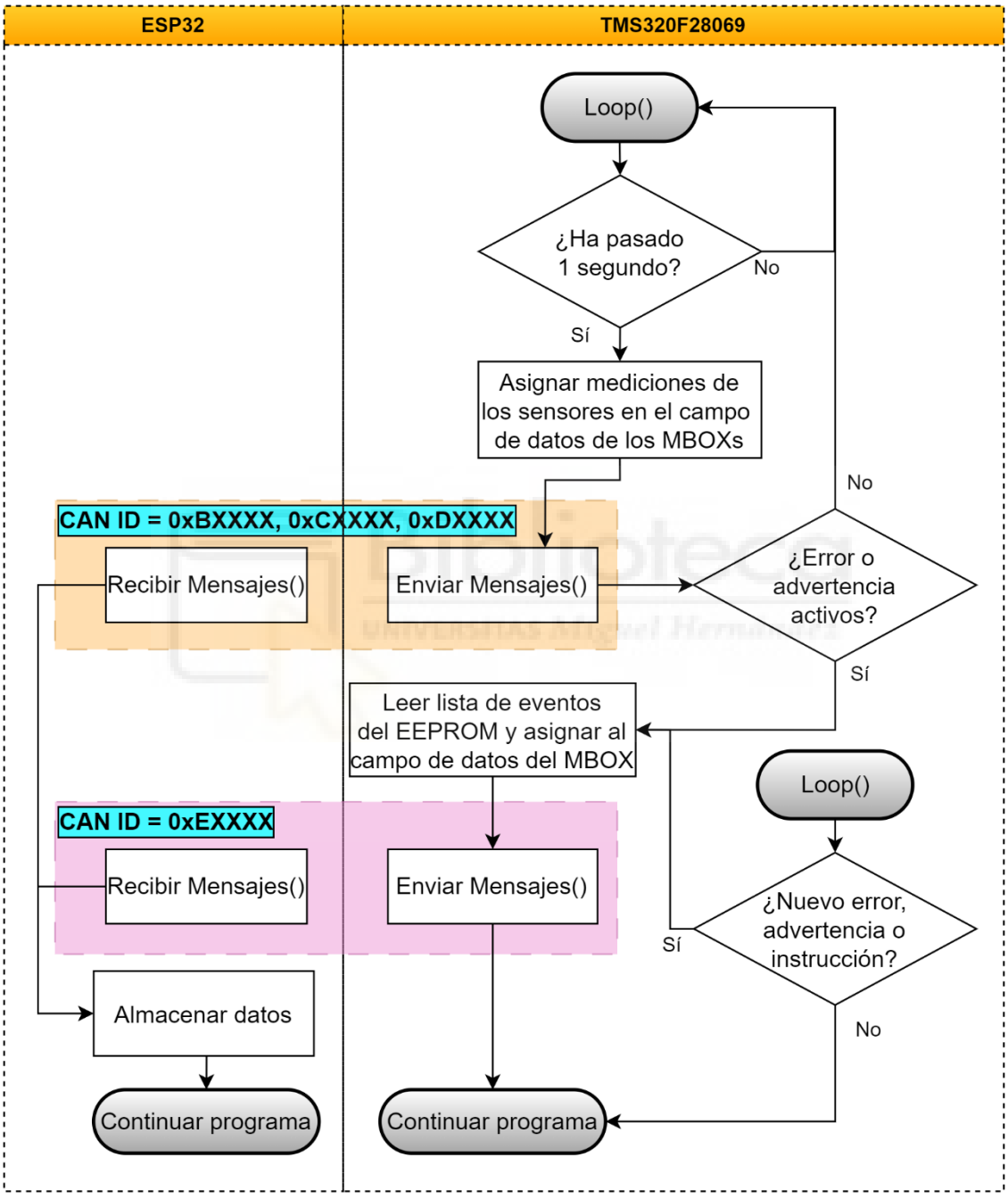

Figura 2.41. Diagrama del proceso de envío de datos y eventos

El dispositivo USG758 almacena unos registros para los errores y advertencias en tiempo real, y en la EEPROM hay una página conocida como lista de eventos dedicada a

almacenar el historial de estos eventos y su marca temporal. Si el bit de alguno de estos registros está activo, se enviará la lista de eventos cada segundo mientras no se solucione. También se enviará la lista de eventos cuando se genere una nueva advertencia, error o instrucción (p.ej. magnetizar), independientemente de si hay un error o advertencia activo. Como la lista de eventos se almacena en una página de la EEPROM, hemos reutilizado parte del código. Cuando se va a enviar la lista de eventos, se lee y almacena esta página en `CAN_Daten[]` y se envía la página de la EEPROM, excepto sin enviar el checksum y el número de página.

El ESP32 irá recibiendo y almacenando la lista como se muestra en la Figura 2.42:

```
203  void receiveWarnungFehler(void){
204      static int BytePtr = 0;
205
206      for(int i=0; i<8; i++){
207          EEprom[PAGE_6].Daten[i+BytePtr*8] = rxFrame.data[i];
208
209      }
210      BytePtr++;
211      if(BytePtr == 32 || BytePtr < 0){
212          WarnungFehlerReceivedFlag = true;
213          BytePtr = 0;
214      }
215  }
```

Figura 2.42. Fragmento de código – Almacenamiento de la lista de eventos en el ESP32

Ya hemos desgranado casi todo el código en partes más detalladas, excepto un par de cosas. Todavía no hemos explicado que sucede en las funciones 'Enviar Mensaje()' y 'Recibir Mensaje()' detalladamente, y se trata de las funciones más básicas para que suceda la comunicación CAN entre ambos microcontroladores.

En el siguiente diagrama podemos observar las acciones que realizan ambos microcontroladores para enviar y recibir un mensaje.

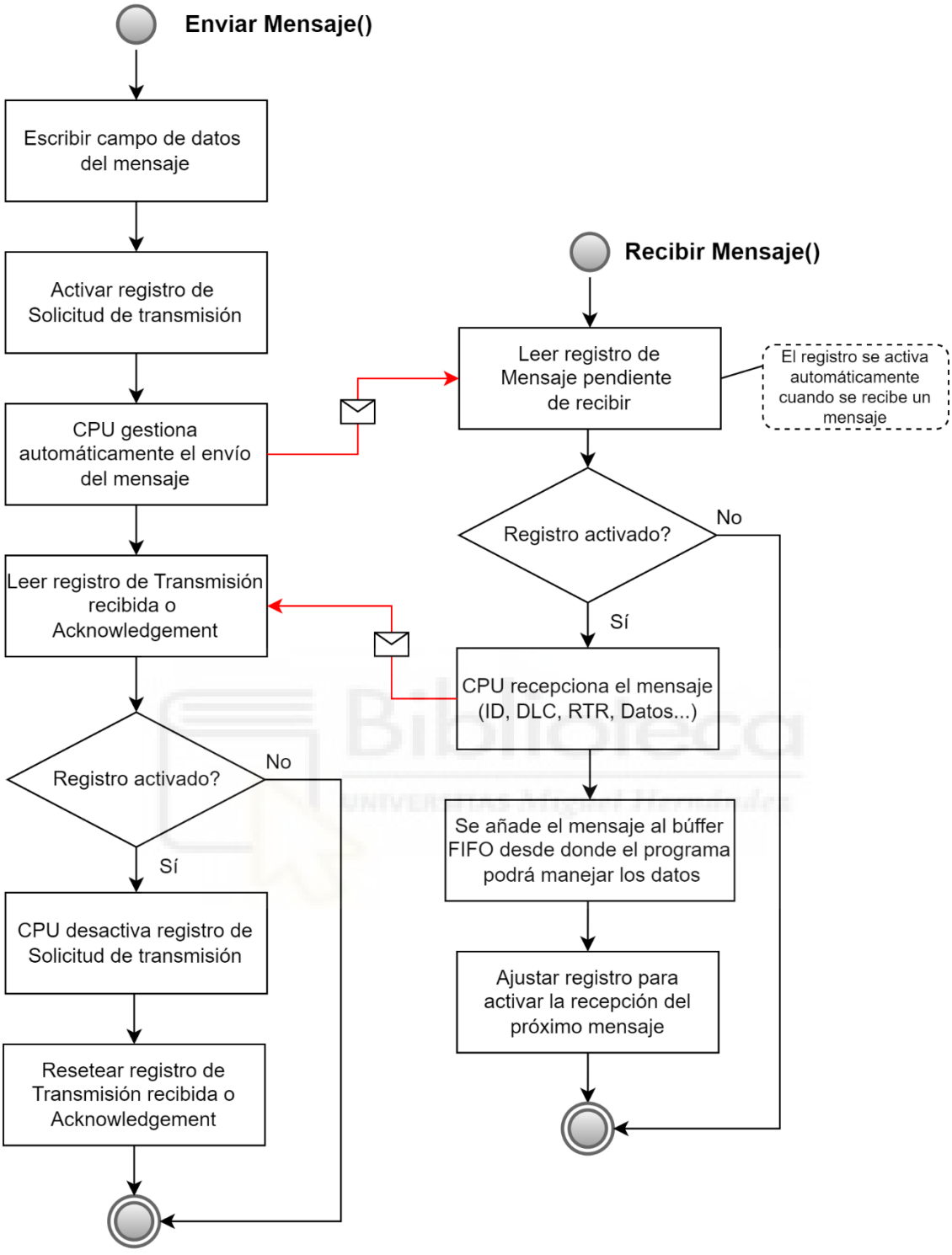

Figura 2.43. Diagrama del proceso de transmisión/recepción de un mensaje

Se debe aclarar que cada microcontrolador tiene un controlador CAN diferente y hay alguna pequeña variación entre el diagrama y lo que ocurre realmente. El diagrama mostrado se ha obtenido a partir de la documentación técnica de ambos microcontroladores. Naturalmente, aunque ambos microcontroladores tengan registros

con la misma finalidad, el nombre de los registros difiere de un microcontrolador a otro. En el TMS320 además se ha implementado al transmitir que si nadie recibe el mensaje se siga enviando hasta pasados 100ms, como se muestra en la Figura 2.44, ya que en el Manual técnico [13] se recomienda enviar constantemente el mensaje mientras no haya un receptor.

```
228         // Transmit starts
229         ECanaShadow.CANTRS.all = 0;
230         ECanaShadow.CANTRS.bit.TRS10 = 1;
231         ECanaRegs.CANTRS.all = ECanaShadow.CANTRS.all;
232         CANTimerACKtimeout = 100;
233             do {
234                 ECanaShadow.CANTA.all = ECanaRegs.CANTA.all;
235                 if(CANTimerACKtimeout == 0) break;
236             }   // WAIT ACKNOWLEDGE
237             while(ECanaShadow.CANTA.bit.TA10 == 0);
238 
239         ECanaShadow.CANTA.all = 0;                  // RESET REGISTER
240         ECanaShadow.CANTA.bit.TA10 = 1;
241         ECanaRegs.CANTA.all = ECanaShadow.CANTA.all;
```

Figura 2.44. Fragmento de código – Transmisión de un mensaje en el TMS320

Con este último diagrama, se ha mostrado el flujo completo del programa desde la inicialización, explicado detalladamente.

3 RESULTADOS Y DISCUSIÓN

3.1 PUESTA EN MARCHA

Antes de probar los programas desarrollados en la red CAN, se comprobó que cada nodo funcionara correctamente por separado, lógicamente para descartar posibles errores. Esta puesta en marcha se realizó en pequeños pasos para tener todo bajo control. En primer lugar, se conectó cada microcontrolador con el analizador PCAN-USB, ejecutando programas de ejemplo. Estos programas de ejemplo o parte de ellos tenían el objetivo de probar el envío y recepción de mensajes y la gestión de sus datos.

Una vez que los dispositivos funcionaran por separado, el siguiente paso fue conectarlos y establecer una comunicación básica entre ellos. Se probó el funcionamiento de los MBOXs, registros, identificadores, etc. Tras esto se probaron algunas funcionalidades extra, como el filtro de aceptación de mensajes.

Antes de empezar a escribir código para el programa principal, había que premeditar las necesidades y requisitos futuros. En este proceso es donde se decidió que manera era la más adecuada para, por ejemplo, enviar la EEPROM, que se trata de una lista de datos muy extensa para enviar a través del CAN bus. También se decidió que estructura tendría el identificador, en función del número de dispositivos máximo previsto y de los mailboxes necesarios.

Luego se comenzó a desarrollar y probar el código mostrado en los diagramas, parte por parte, hasta tener todo el código acabado. Durante el desarrollo del código se realizaron varias modificaciones importantes en el hardware y en el software para ir afinando lo más posible los detalles: se probaron redes con más nodos, con diferentes Baud Rates y hasta se inspeccionó un poco en código de bajo nivel para comprender algunas funciones.

La principal aplicación de la comunicación en sí es que el usuario controle el dispositivo USG758 a través de la consola, por esto es interesante poder ver algún ejemplo de ello. Entonces se ha creado un menú de usuario simple y provisional que permite esta interacción con el usuario.

Al principio la pantalla muestra un menú principal, y a través de la ruleta o encoder rotatorio y los botones podemos navegar, seleccionar o volver atrás en el menú. Las opciones principales del menú son las mostradas en la Figura 3.1:

Figura 3.1. Menú principal de la consola

Cada opción del menú principal nos lleva a un menú secundario, en el cual podemos observar datos o guardar nuestra selección para la configuración. Algunos de estos submenús y sus opciones se muestran en la Figura 3.2.

Figura 3.2. Submenús de la consola

Por ejemplo, en el submenú “Tipo Iman” podemos cambiar que tipo de imán estamos usando. En el submenú “Leer EEprom” podemos solicitar la lectura del EEprom. En el submenú “Datos sensores” podemos ver las medidas del USG758 actualizarse cada segundo. En el submenú “Eventos” podemos observar la lista de acciones, advertencias y errores.

En conjunto, este menú nos permite poner en acción todas las funcionalidades que hemos diseñado a través del CAN bus y ver una pequeña implementación práctica de cómo el usuario será capaz de controlar el dispositivo USG758 a través de la placa de usuario y del CAN bus.

3.2 ANALISIS DE LA COMUNICACIÓN

En este apartado veremos la concordancia de los resultados observables desde el osciloscopio y el analizador PCAN-USB.

Desde la aplicación PCAN-View podemos observar los datos registrados por el analizador PCAN-USB. Estos datos son el identificador, el bit DLC, los datos, el intervalo de tiempo entre mensajes en milisegundos y el recuento de mensajes.

Dentro de la aplicación PicoScope 6, en la parte superior podemos observar la señal del osciloscopio. Hemos monitoreado la señal CANH (en azul) y CANL (en rojo). La señal en morado es la señal diferencial (CANH-CANL). Justo debajo de la señal morada se encuentra el decodificador, que traduce las ondas en datos visibles para nosotros. En la parte inferior de la ventana se encuentra la tabla de decodificación, que muestra los datos de las tramas CAN de hasta las últimas 32 mediciones.

En la Figura 3.3 y la Figura 3.4 podemos observar el bus CAN cuando se envían los datos de sensores cíclicamente cada segundo (Comando del ID = B, C, D):

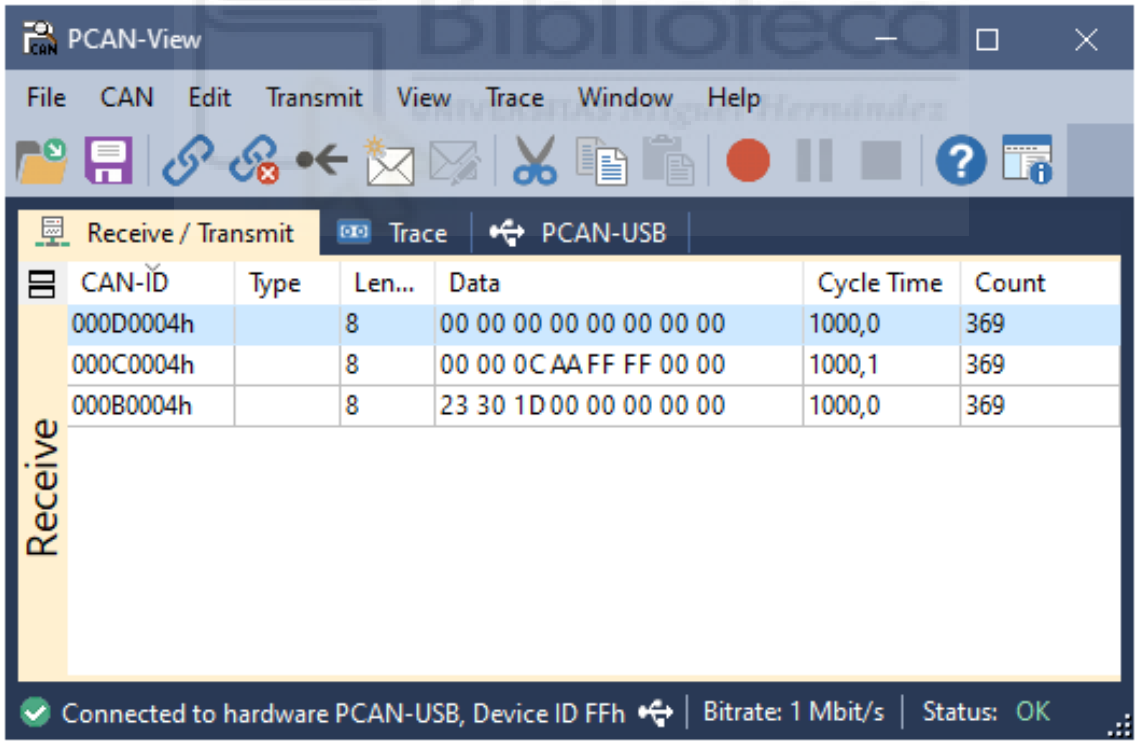

Figura 3.3. Captura de PCAN-View cuando se envían los datos cíclicos

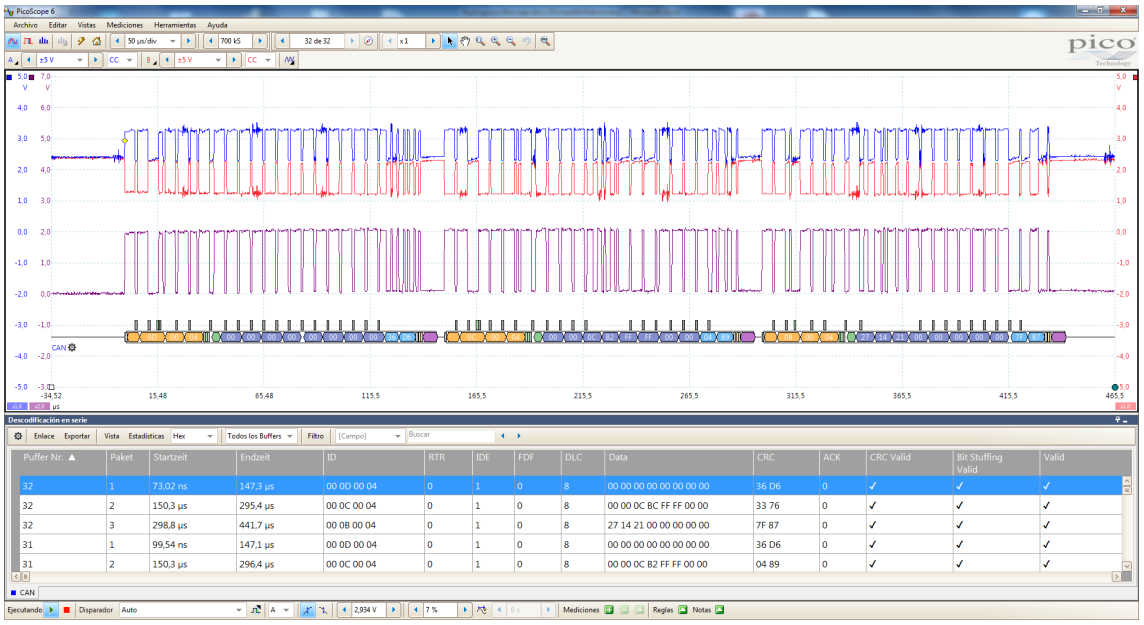

Figura 3.4. Captura del osciloscopio cuando se envían los datos cíclicos

ID	RTR	IDE	FDF	DLC	Data	CRC
00 0D 00 04	0	1	0	8	00 00 00 00 00 00 00 00	36 D6
00 0C 00 04	0	1	0	8	00 00 0C BC FF FF 00 00	33 76
00 0B 00 04	0	1	0	8	27 14 21 00 00 00 00 00	7F 87
00 0D 00 04	0	1	0	8	00 00 00 00 00 00 00 00	36 D6
00 0C 00 04	0	1	0	8	00 00 0C B2 FF FF 00 00	04 89

Figura 3.5. Zoom de la Figura 3.4 sobre la tabla de descodificación

En la Figura 3.6 y la Figura 3.7 se muestra el bus justo tras seleccionar una opción desde el menú de la consola, en este caso, el envío de la EEPROM (Comando del ID = 4). Cuando se selecciona una opción, la consola manda un mensaje con los parámetros de las variables. Cada una de estas variables hace referencia a una orden o configuración. El byte 3 (cuyo valor es 02) almacena la variable que activa el envío de la EEPROM. Cada vez que se envía este mensaje, el USG758 compara su valor con el que hemos programado, y si coincide, se enviará la EEPROM.

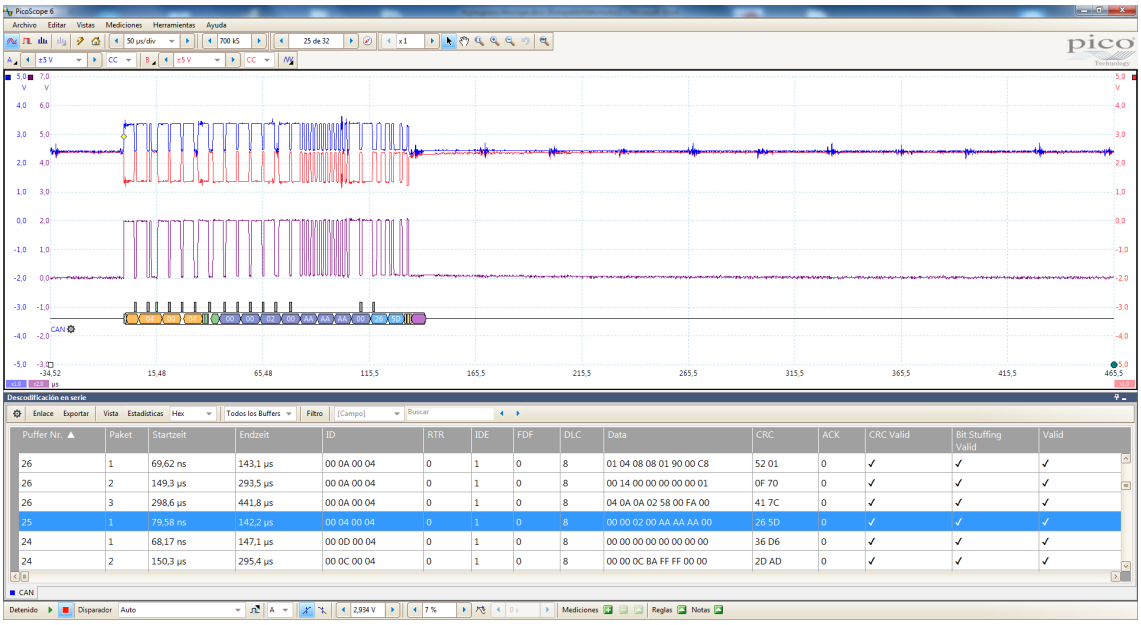

Figura 3.6. Captura del osciloscopio cuando se envía la solicitud de EEPROM

ID	RTR	IDE	FDF	DLC	Data	CRC
00 0A 00 04	0	1	0	8	01 04 08 08 01 90 00 C8	52 01
00 0A 00 04	0	1	0	8	00 14 00 00 00 00 00 01	0F 70
00 0A 00 04	0	1	0	8	04 0A 0A 02 58 00 FA 00	41 7C
00 04 00 04	0	1	0	8	00 00 02 00 AA AA AA 00	26 5D
00 0D 00 04	0	1	0	8	00 00 00 00 00 00 00 00	36 D6
00 0C 00 04	0	1	0	8	00 00 0C BA FF FF 00 00	2D AD

Figura 3.7. Zoom de la Figura 3.6 sobre la tabla de descodificación

Y después de la solicitud de la memoria EEPROM, el USG758 la envía. Recordemos que la EEPROM completa ocupa 198 mensajes (33 mensajes por página * 6 páginas). Además, hemos programado que se envíen los mensajes sin demora, por tanto, podemos observar como el “Cycle Time” o intervalo entre mensajes es 0,2ms que es el valor esperado a 1Mbit/s. En las siguientes figuras podemos observar la señal cuando se envía la EEPROM (Comando del ID = A):

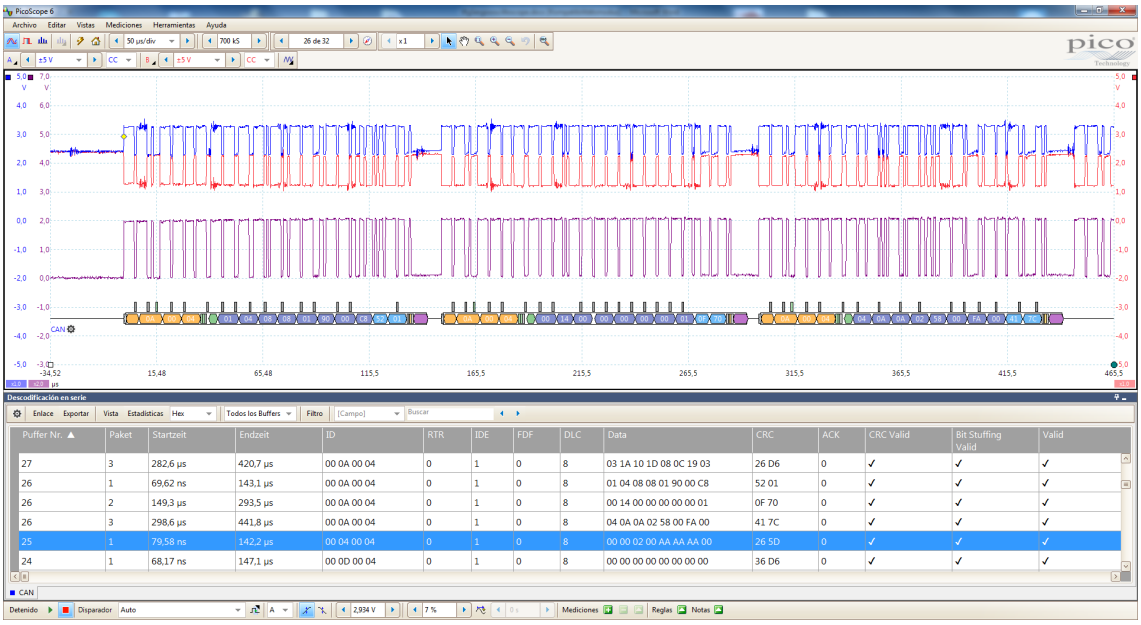

Figura 3.8. Captura del osciloscopio cuando se envía el contenido de la EEPROM

ID	RTR	IDE	FDF	DLC	Data	CRC
00 0A 00 04	0	1	0	8	03 1A 10 1D 08 0C 19 03	26 D6
00 0A 00 04	0	1	0	8	01 04 08 08 01 90 00 C8	52 01
00 0A 00 04	0	1	0	8	00 14 00 00 00 00 00 01	0F 70
00 0A 00 04	0	1	0	8	04 0A 0A 02 58 00 FA 00	41 7C
00 04 00 04	0	1	0	8	00 00 02 00 AA AA AA 00	26 5D
00 0D 00 04	0	1	0	8	00 00 00 00 00 00 00 00	36 D6

Figura 3.9. Zoom de la Figura 3.8 sobre la tabla de descodificación

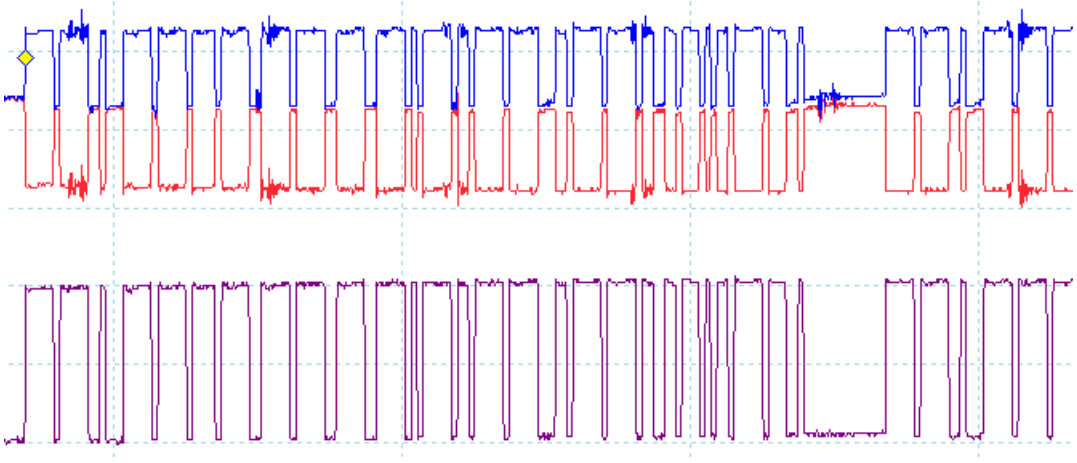

Figura 3.10. Zoom de la señal del osciloscopio de la Figura 3.8

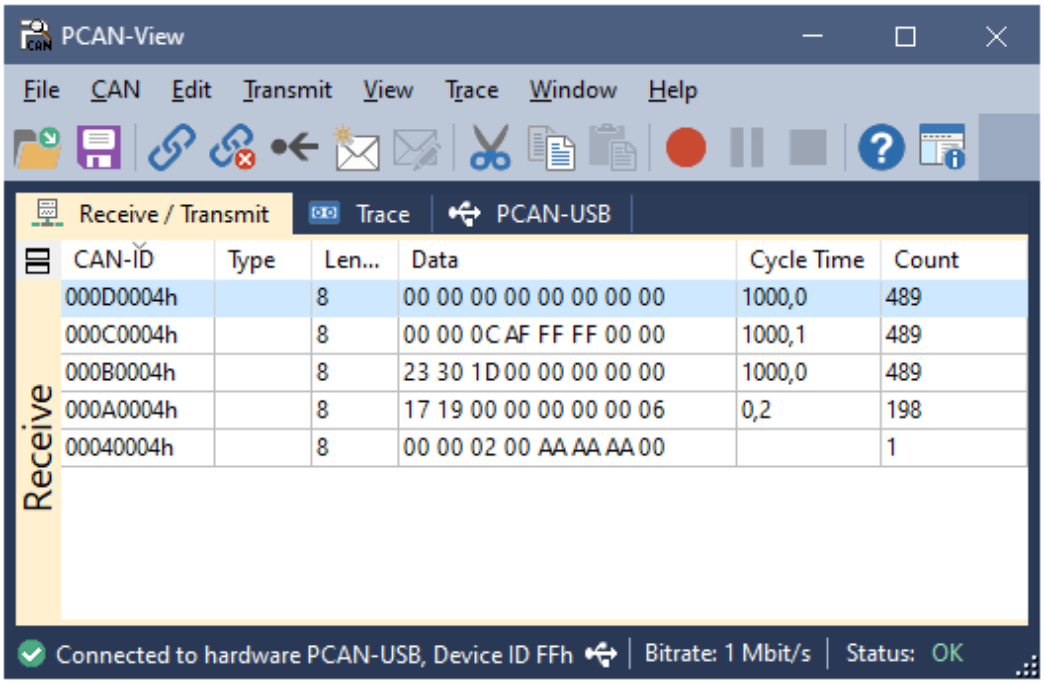

Figura 3.11. Captura de PCAN-View cuando se envía la EEPROM

Como podemos observar en la señal del osciloscopio en la Figura 3.10, hay interferencias en las señales CANH y CANL constantemente debido a interferencias electromagnéticas, originadas probablemente por el USG758. Pero en la señal diferencial se suprimen y se ve una señal completamente plana, entonces no influyen en la calidad de la transmisión. Los niveles de voltaje son correctos, CANH va desde 2,25 a 3,3V y CANL va desde 1,2 a 2,25V aproximadamente. El punto medio entre 1,2 y 3,3V es 2,25V, como se espera. Desde PCAN-View podemos ver que se envían el número exacto de mensajes con el periodo exacto que se ha programado.

Por tanto, además de realizarse la comunicación, las mediciones de los aparatos de medida son aparentemente correctas.

Cabe mencionar que si el analizador PCAN-USB no está en modo “solo escucha”, se puede observar en el osciloscopio un pequeño pico en la señal en el bit de reconocimiento o ACK. Esto es porque el analizador tiene un voltaje un poco más alto, y al responder con el bit de reconocimiento, se aprecia este pequeño pico de voltaje. Esto no supone ningún problema en la comunicación.

Otro detalle a mencionar es que si se observa la señal del osciloscopio, cuando se envían los datos de los sensores, los mensajes no siguen el orden de prioridad del sistema de arbitraje (el mensaje con ID más pequeño prevalece). Esto es porque el controlador CAN

del TMS320 tiene incorporado un sistema independiente de prioridad en los MBOXs de transmisión. Cuando varios MBOXs están listos para transmitir un mensaje y tienen el mismo nivel de prioridad, se transfiere primero el mensaje del MBOX de mayor numeración. El nivel de prioridad del MBOX es modificable desde el registro MSGCTRL. Para el proyecto no hemos usado este sistema de prioridad, pero está bien conocer su funcionamiento.

3.3 PROBLEMAS Y SOLUCIONES

PROBLEMA 1 – Transductor averiado

El USG758 no era capaz de enviar ninguna señal a través del bus y no sabíamos por qué. No se detectaba ningún intento de enviar nada al bus a través del osciloscopio. Tras una búsqueda exhaustiva en el software, estaba todo en condiciones. También se comprobó que las pistas de la placa electrónica estuvieran bien conectadas, midiendo entre pines mediante la prueba de continuidad con el multímetro y que el transductor estuviera alimentado correctamente. Tras esto probamos con un programa a modular la señal de salida directamente del puerto del chip TMS320, por si fuera problema del microcontrolador, y funcionaba correctamente.

Todo esto nos indicaba que el problema estaba por descarte en el transductor. Cambiamos el transductor por uno nuevo y funcionó correctamente. No estamos completamente seguros, pero es posible que se estropeara debido a una conexión incorrecta de los puertos que provocara la rotura de algún componente interno del transductor. Esto nos retrasó un par de días. Mostrando este problema, se pretende reflejar una parte de la metodología que conlleva el trabajo con componentes electrónicos, como es la búsqueda del origen de los fallos.

PROBLEMA 2 – Bit Timing

Cuando se comprobó que cada nodo funcionaba correctamente por separado, hubo complicaciones con los parámetros de Bit Time, en concreto con el TMS320.

La complicación estuvo en encontrar y entender la frecuencia del reloj del microcontrolador, ya que, con nula experiencia en este tipo de dispositivos, hay una

pequeña barrera de entrada en el manejo de sus librerías o proyectos, que suelen tener un tamaño considerable. Además, la frecuencia del reloj se definía en función de varios parámetros, cuyo cálculo no estaba claramente explicado. El USG758 cuenta con un Reloj externo (RTC) incorporado, que también complicó un poco las cosas al pasar de los programas de ejemplo al del proyecto. Conocer esta frecuencia del reloj del microcontrolador es vital para la configuración del Bit Time, por ello es muy importante conocer su manejo.

PROBLEMA 3 – Cambio de biblioteca

En un principio, había optado por utilizar la biblioteca Arduino CAN de Sandeep Mistry [18] para el ESP32, debido a que era relativamente más sencilla que otras librerías. Pero tras ponerla en marcha con alguna pequeña dificultad e incluso haber avanzado gran parte del código, observé un problema extraño: si se recibían mensajes con un intervalo de 15 milisegundos o menos entre cada uno, había mensajes que no se recibían correctamente. Esto no era problema del bus, ya que el analizador CAN captaba todo correctamente. Tras indagar el comportamiento de los registros y en internet, encontré que el fallo provenía del buffer FIFO: Data Overrun. El buffer se desbordaba ya que no era capaz de procesar todos los mensajes con suficiente velocidad. El buffer FIFO tiene capacidad para 64 bytes, y cada mensaje contiene 3-13 bytes, por tanto, cuando se envían 5 mensajes de 13 bytes seguidos llegan 65 bytes, y el último byte no se almacena correctamente causando posibles desplazamientos. Tras probar algunas posibles soluciones, como ejecutar algunas funciones desde la RAM interna, logré reducir a 2ms el intervalo mínimo de recepción mientras funcionara correctamente. Pero seguía sin ser lo correcto, ya que a 1Mbit/s cada mensaje puede transmitirse cada 150 µs (1 bit/µs y cada mensaje tiene entre 100-150 bits normalmente). Opté por probar la biblioteca ESP32-TWAI-CAN, que usa directamente el driver TWAI, y en esta funcionaba correctamente. Para ello realicé un programa aislado con el objetivo de comprobar que los mensajes se reciban a la velocidad adecuada, el cuál podemos observar en la figura:

```
213  void loop() {
246      if (ESP32Can.readFrame(rxFrame, 0)) {
248          RxQ[cnt] = ESP32Can.inRxQueue();
249          tiempo[cnt] = currentStamp - lastStamp;
250          lastStamp = currentStamp;
251          cnt++;
252          if(cnt > 197){
253              cnt = 0;
254              for(int i = 0; i < 198; i++){
255                  Serial.printf("RxQ[%d]: %d\t", i, RxQ[i]);
256                  Serial.printf("tiempo[%d]: %d\n", i, tiempo[i]);
257              }
258          }
259      }
260
```

Figura 3.12. Programa para registrar el tiempo de recepción entre mensajes

El funcionamiento del programa es: Cuando se recibe un mensaje, almacenar cuantos mensajes hay pendientes de leer desde el búffer FIFO (`RxQ[]`) y cuanto tiempo pasó desde que se recibió el último mensaje en microsegundos (`tiempo[]`). Cuando se hayan recibido 198 mensajes, mostrar los datos almacenados. Es importante no mostrar los datos almacenados uno a uno tras cada mensaje, ya que el tiempo necesario para mostrar un mensaje por consola ronda los 5ms.

Los resultados del programa tras forzar el envío de la EEPROM se muestra a continuación en la Figura 3.13.

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL

CAN Receiver
CAN bus started!
RxQ[0]: 0       tiempo[0]: 12649642
RxQ[1]: 0       tiempo[1]: 6775
RxQ[2]: 0       tiempo[2]: 149
RxQ[3]: 0       tiempo[3]: 144
RxQ[4]: 0       tiempo[4]: 144
RxQ[5]: 0       tiempo[5]: 144
RxQ[6]: 0       tiempo[6]: 139
RxQ[7]: 0       tiempo[7]: 140
RxQ[8]: 0       tiempo[8]: 160
RxQ[9]: 0       tiempo[9]: 140
RxQ[10]: 0      tiempo[10]: 155
RxQ[11]: 0      tiempo[11]: 149
RxQ[12]: 0      tiempo[12]: 140
RxQ[13]: 0      tiempo[13]: 150
RxQ[14]: 0      tiempo[14]: 161
RxQ[15]: 0      tiempo[15]: 142
RxQ[16]: 0      tiempo[16]: 163
RxQ[17]: 0      tiempo[17]: 148
RxQ[18]: 0      tiempo[18]: 143
RxQ[19]: 0      tiempo[19]: 160
RxQ[20]: 0      tiempo[20]: 150
RxQ[21]: 0      tiempo[21]: 153
RxQ[22]: 0      tiempo[22]: 151
RxQ[23]: 0      tiempo[23]: 142
RxQ[24]: 0      tiempo[24]: 161
RxQ[25]: 0      tiempo[25]: 150
RxQ[26]: 0      tiempo[26]: 143
RxQ[27]: 0      tiempo[27]: 161
RxQ[28]: 0      tiempo[28]: 149
RxQ[29]: 0      tiempo[29]: 154
RxQ[30]: 0      tiempo[30]: 141
RxQ[31]: 0      tiempo[31]: 161
RxQ[32]: 0      tiempo[32]: 151
RxQ[33]: 0      tiempo[33]: 222
RxQ[34]: 0      tiempo[34]: 26636
RxQ[35]: 0      tiempo[35]: 158
RxQ[36]: 0      tiempo[36]: 140
RxQ[37]: 0      tiempo[37]: 151
RxQ[38]: 0      tiempo[38]: 158
RxQ[39]: 0      tiempo[39]: 142
RxQ[40]: 0      tiempo[40]: 161
```

Figura 3.13. Resultados del registro de recepción entre mensajes.

Como podemos ver, cada mensaje se recibe cada 150 µs aproximadamente, valores correctos para una transmisión a 1Mbit/s. En la medida `RxQ[34]` se envía el checksum con una espera de 20 ms, y además en el ESP32 se muestra por pantalla el checksum, que requiere de 5ms aproximadamente. Por esto aparece un tiempo de 26 ms.

Esta biblioteca funciona correctamente debido a que usa funciones con la RAM interna y el driver TWAI usa manejadores de funciones (handlers) que optimizan mucho más el código. También incluye funciones que permiten modificar el buffer de recepción y transmisión de mensajes, cosa que la anterior biblioteca no facilitaba.

PROBLEMA 4 – Terminación con más nodos

Aunque este problema no ha influido directamente en el proyecto, debería tenerse en cuenta en el futuro. La red CAN será normalmente de un número indefinido de nodos por ser ampliable, pero siempre habrá dos nodos que deben tener en su interior la resistencia de terminación de 120Ω. Podemos suponer que la consola siempre tendrá una resistencia de terminación. Pero como los dispositivos USG758 son supuestamente iguales tras la producción, no se puede tener dos resistencias de terminación ya que tendrían esta resistencia o todos o ningún dispositivo.

Por ello, una posible solución a esto sería añadir en cada dispositivo USG758 una resistencia, que sea configurable desde el exterior mediante un interruptor o software. El inconveniente de esto es que cualquier persona podría modificar la resistencia sin saber su función necesariamente. Entonces los clientes deberán conocer como configurar las resistencias de terminación del bus CAN.

Otra alternativa, restringiendo el acceso a las personas externas, sería según la instalación del cliente soldar una resistencia de terminación en la placa electrónica del último dispositivo y dejarlo fijo. El inconveniente de esta opción es que cuesta más esfuerzo cada vez que se quiera conectar o desconectar esta resistencia, pero sigue siendo muy buena opción si no se modificará la topología de la red con frecuencia.

4 CONCLUSIONES

4.1 CONCLUSIONES OBTENIDAS

En el presente TFG se ha llevado a cabo la implementación práctica de una red de comunicación CAN entre dos microcontroladores diferentes. Los chips han sido utilizados para controlar la consola o interfaz de usuario y el dispositivo USG758. El objetivo principal, el cuál era implementar esta red CAN diseñada para gestionar el dispositivo USG758 desde la consola e intercambiar datos en tiempo real, ha sido completado con éxito.

A lo largo del desarrollo del proyecto, se han realizado configuraciones de hardware y software, pruebas de transmisión y análisis de la comunicación para garantizar una comunicación segura entre los dispositivos. Los pocos problemas que han surgido se deben principalmente a la complejidad técnica de adaptarse a un nuevo hardware y software.

Se ha mostrado varios ejemplos de utilización típicos para el CAN bus, como puede ser la transmisión de datos de sensores o el envío de varios mensajes que componen una base de datos mayor. También se han creado pequeños ejemplos interactivos para el usuario a través de un menú.

Asimismo, se han cumplido todos los objetivos secundarios, excepto uno, controlar varios dispositivos USG758 desde una sola consola. A pesar de que se ha implementado código para ello, no se ha probado ya que solo se contaba con un dispositivo.

En cuanto a mi desarrollo personal, considero que he adquirido un conocimiento específico sobre una tecnología ampliamente usada en la industria como es el CAN bus. Esto es enriquecedor para mi futuro profesional y será de gran ayuda a la hora de trabajar con otros protocolos de comunicación. Además, he ampliado enormemente mis conocimientos sobre microcontroladores al trabajar mano a mano con dos modelos diferentes.

4.2 LINEAS FUTURAS

Como líneas futuras se plantean varias alternativas:

- Continuar el desarrollo de la consola para que sea completamente funcional como interfaz de usuario, es decir, crear un menú, asignar que funciones podrá o no modificar el usuario, que controles deberá usar, etc. Así la consola podría ofrecerse como un producto adicional para controlar los dispositivos USG758.
- Otro futuro proyecto podría ser implementar que desde un dispositivo USG758 se pueda controlar los otros dispositivos, a través de CAN bus. Si podemos controlar varios dispositivos desde una sola placa frontal, se puede ahorrar el gasto de fabricar una placa frontal para cada dispositivo y simplificaría la producción de los dispositivos USG758.
- Un desarrollo adicional podría ser la implementación de un servidor web desde el ESP32 para configurar los dispositivos USG758 u observar datos en tiempo real. El ESP32 cuenta con un módulo Wifi que hace esto posible. Así el usuario podría monitorear o modificar parámetros no solo desde la consola, sino desde el móvil o el ordenador sin necesidad de cables.

Gracias a la implementación del CAN bus se abre un abanico de posibilidades en el futuro desarrollo del USG758 o la consola.

5 BIBLIOGRAFÍA

[1] Wagner Magnete GmbH & Co. KG, «Wagner Magnete Webpage,» https://www.wagner-magnete.com/de/. [Último acceso: Marzo 2025].

[2] International Organization for Standardization, «ISO 11898-1:2024,» https://www.iso.org/standard/86384.html. [Último acceso: Enero 2025].

[3] AutoPi.io, «https://www.autopi.io/blog/can-bus-explained/,».

[4] Wikipedia, «Bus CAN,» https://es.wikipedia.org/wiki/Bus_CAN. [Último acceso: Diciembre 2024].

[5] Texas Instruments Inc., «ISO1050 Isolated CAN Transceiver Datasheet,» https://www.ti.com/lit/ds/symlink/iso1050.pdf.

[6] A. B. Florian Hartwich, «The Configuration of the CAN Bit Timing,» de *6th International CAN Conference*, Turin (Italy).

[7] M. Falch, «CAN Bus Errors Explained - A Simple Intro,» CSS Electronics, https://www.csselectronics.com/pages/can-bus-errors-intro-tutorial. [Último acceso: Febrero 2025].

[8] Espressif Systems, «ESP32 Series Datasheet,» https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. [Último acceso: Enero 2025].

[9] Texas Instruments Inc., «TMS320F28069,» https://www.ti.com/product/TMS320F28069. [Último acceso: Abril 2025].

[10] Texas Instruments Inc., «TMS320F2806x Real-Time Microcontrollers Datasheet,» https://www.ti.com/lit/ug/spruh18i/spruh18i.pdf?ts=1740431341280. [Último acceso: Enero 2025].

[11] M. s. L. (handmade0octopus), «ESP32-TWAI-CAN Library,» https://github.com/handmade0octopus/ESP32-TWAI-CAN.git. [Último acceso: Marzo 2025].

[12] Espressif Systems (Shanghai) Co., Ltd., «Two-Wire Automotive Interface (TWAI) API,» https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/twai.html. [Último acceso: Marzo 2025].

[13] Espressif Systems (Shanghai) Co., Ltd, «ESP32 Technical Reference Manual V5.3,» https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf. [Último acceso: Febrero 2025].

[14] H. Janakiraman, «Programming Examples for the TMS320x28xx eCAN,» Enero 2003. https://www.ti.com/lit/an/spra876b/spra876b.pdf. [Último acceso: Febrero 2025].

[15] PlatformIO, «Platformio Quick Start,» https://docs.platformio.org/en/latest/integration/ide/vscode.html#quick-start. [Último acceso: Abril 2025].

[16] Texas Instruments Inc., «2833x/2823x C/C++ Header Files and Peripheral Examples Quick Start,» https://www.ti.com/lit/ml/sprca73/sprca73.pdf. [Último acceso: Abril 2025].

[17] A. Haun, «Calculator for CAN Bit Timing Parameters,» Texas Instruments, https://www.ti.com/lit/an/sprac35/sprac35.pdf. [Último acceso: Febrero 2025].

[18] S. Mistry, «Arduino CAN library,» https://github.com/sandeepmistry/arduino-CAN. [Último acceso: Enero 2025].

6 GLOSARIO

Sistema embebido: sistema compuesto por una combinación de hardware y software diseñado para realizar una tarea específica de forma automática.

Consola: dispositivo que, integrado o no en una máquina, contiene los instrumentos para su control y operación.

Microcontrolador: computadora pequeña construida en un pequeño chip.

Pin, GPIO, Puerto: punto físico de conexión en un dispositivo electrónico que permite la entrada o salida de señales eléctricas.

Protocolo de comunicación: conjunto de normas o instrucciones que sirven para guiar las acciones durante el intercambio de información.

CAN (Controller Area Network): protocolo de comunicación entre dispositivos sin necesidad de un ordenador central, especialmente utilizado en automoción.

Nodo: punto de conexión dentro de una red capaz de enviar, recibir y procesar información.

Baud rate (tasa de baudios): define cómo de rápido son transmitidos los datos en una comunicación en serie.

Arbitraje: Mecanismo utilizado en redes de comunicación para decidir que dispositivo transmite primero cuando varios intentan enviar datos al mismo tiempo.

Frame, Trama: en redes se refiere a una unidad de envío de datos.

ACK o acknowledgement (reconocimiento): señal de confirmación que indica la recepción exitosa de una trama.

Transductor: dispositivo que convierte energía de una forma a otra. En nuestro contexto, transforma señales de forma digital a analógica y viceversa.

Filtro o máscara de aceptación: mecanismo que decide que mensajes son aceptados según un criterio.

MBOX, Mailbox (buzón): Depósito o memoria donde se almacenan y organizan los mensajes recibidos.

Biblioteca: en software, colección de funciones o rutinas que facilitan el desarrollo de programas.

Driver (controlador): programa informático que facilita o permite la interacción entre el sistema operativo y el hardware o periféricos.

Debugger (depurador): herramienta de software o hardware que permite supervisar, controlar y modificar la ejecución del código para encontrar posibles errores.

Bit time, Bit timing: intervalo de tiempo durante el cual se transmite un único bit de información.

ID o identificador: valor único asignado a un dispositivo o mensaje con el propósito de diferenciarlo de los demás.

EEPROM (Electrically Erasable Programmable Read-Only Memory): memoria que se puede borrar y reprogramar mediante señales eléctricas.

Checksum (suma de verificación): valor numérico calculado a partir de los datos de un mensaje, con el objetivo de detectar posibles errores durante el envío o procesado de los datos.

Registro: en arquitectura de ordenadores, pequeña unidad de almacenamiento de alta velocidad utilizada para guardar temporalmente datos, direcciones o instrucciones durante la ejecución de programas.

Bit: unidad de información más pequeña, que representa dos estados posibles: 0 o 1.

Byte: unidad básica de información, formada por 8 bits.

Buffer: sector de memoria digital reservado para el almacenamiento temporal de información aún no procesada.

7 ANEXO DE CÓDIGO

Algunas partes mínimas del código se han omitido por motivos de confidencialidad y puede haber variables no definidas correctamente, ya que se ha agrupado código de diferentes archivos para simplificar el código y evitar repeticiones innecesarias. Alguna parte del código puede estar sujeta a copyright.

7.1 TMS320F28069

INICIALIZACIÓN MÓDULO CAN:

```
// TI File $Revision: /main/8 $
// Checkin $Date: June 25, 2008   15:19:07 $
//###########################################################################
//
// FILE:    DSP2833x_ECan.c
//
// TITLE:   DSP2833x Enhanced CAN Initialization & Support Functions.
//
//###########################################################################
// $TI Release: 2833x/2823x Header Files V1.32 $
// $Release Date: June 28, 2010 $
// $Copyright:
// Copyright (C) 2009-2024 Texas Instruments Incorporated - http://www.ti.com/
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//
//   Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
//
//   Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the
//   distribution.
//
//   Neither the name of Texas Instruments Incorporated nor the names of
//   its contributors may be used to endorse or promote products derived
//   from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

```
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
// $
//###########################################################################

//
// Included Files
//
#include "F2806x_Device.h"      // F2806x Headerfile Include File
#include "F2806x_Examples.h"    // F2806x Examples Include File


struct ECAN_REGS ECanaShadow;

void InitECan(void)
{
    InitECana();
}

//
// InitECana - Initialize eCAN-A module
//
void InitECana(void)
{
    //
    // Create a shadow register structure for the CAN control registers. This
    // is needed, since only 32-bit access is allowed to these registers.
    // 16-bit access to these registers could potentially corrupt the register
    // contents or return false data. This is especially true while writing
    // to/reading from a bit (or group of bits) among bits 16 - 31
    //

    EALLOW;     // EALLOW enables access to protected bits

    //
    // Configure eCAN RX and TX pins for CAN operation using eCAN regs
    //
    ECanaShadow.CANTIOC.all = ECanaRegs.CANTIOC.all;
    ECanaShadow.CANTIOC.bit.TXFUNC = 1;
    ECanaRegs.CANTIOC.all = ECanaShadow.CANTIOC.all;

    ECanaShadow.CANRIOC.all = ECanaRegs.CANRIOC.all;
    ECanaShadow.CANRIOC.bit.RXFUNC = 1;
    ECanaRegs.CANRIOC.all = ECanaShadow.CANRIOC.all;

    //
    // Configure eCAN for HECC mode - (reqd to access mailboxes 16 thru 31)
    // HECC mode also enables time-stamping feature
```

```
    //
    ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
    ECanaShadow.CANMC.bit.SCB = 1;
    ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;

    //
    // Initialize all bits of 'Master Control Field' to zero
    // Some bits of MSGCTRL register come up in an unknown state. For proper
    // operation, all bits (including reserved bits) of MSGCTRL must be
    // initialized to zero
    //
    ECanaMboxes.MBOX0.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX1.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX2.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX3.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX4.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX5.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX6.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX7.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX8.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX9.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX10.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX11.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX12.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX13.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX14.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX15.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX16.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX17.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX18.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX19.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX20.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX21.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX22.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX23.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX24.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX25.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX26.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX27.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX28.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX29.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX30.MSGCTRL.all = 0x00000000;
    ECanaMboxes.MBOX31.MSGCTRL.all = 0x00000000;

    //
    // TAn, RMPn, GIFn bits are all zero upon reset and are cleared again
    // as a matter of precaution.
    //
    ECanaRegs.CANTA.all = 0xFFFFFFFF;       // Clear all TAn bits
```

```
    ECanaRegs.CANRMP.all = 0xFFFFFFFF;      // Clear all RMPn bits

    ECanaRegs.CANGIF0.all = 0xFFFFFFFF;     // Clear all interrupt flag bits
    ECanaRegs.CANGIF1.all = 0xFFFFFFFF;

    //
    // Configure bit timing parameters for eCANA
    //
    ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
    ECanaShadow.CANMC.bit.CCR = 1 ;            // Set CCR = 1
    ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;

    ECanaShadow.CANES.all = ECanaRegs.CANES.all;

    do
    {
        ECanaShadow.CANES.all = ECanaRegs.CANES.all;
    } while(ECanaShadow.CANES.bit.CCE != 1 );   // Wait for CCE bit to be set

    ECanaShadow.CANBTC.all = 0;

    //
    // CPU_FRQ_150MHz is defined in DSP2833x_Examples.h
    //
    #if (CPU_FRQ_150MHZ)
    //
    // The following block for all 150 MHz SYSCLKOUT
    // (75 MHz CAN clock) - default. Bit rate = 1 Mbps See Note at End of File
    //
    ECanaShadow.CANBTC.bit.BRPREG = 4;
    ECanaShadow.CANBTC.bit.TSEG2REG = 2;
    ECanaShadow.CANBTC.bit.TSEG1REG = 10;
    #endif

    //
    // CPU_FRQ_100MHz is defined in DSP2833x_Examples.h
    //
    #if (CPU_FRQ_100MHZ)
    //
    // The following block is only for 100 MHz SYSCLKOUT (50 MHz CAN clock).
    // Bit rate = 1 Mbps See Note at End of File
    //
    ECanaShadow.CANBTC.bit.BRPREG = 4;
    ECanaShadow.CANBTC.bit.TSEG2REG = 1;
    ECanaShadow.CANBTC.bit.TSEG1REG = 6;
    #endif

    // Use CAN Bit Timing Calculator App Report from Texas Instruments
    ECanaShadow.CANBTC.all = 0x03022A;          // For 1000 Kbit/s
    //ECanaShadow.CANBTC.all = 0x0F03A3;        // For  250 Kbit/s
```

```
    //ECanaShadow.CANBTC.all = 0x0402A3;        // For 800 Kbit/s

    ECanaShadow.CANBTC.bit.SAM = 1;
    ECanaRegs.CANBTC.all = ECanaShadow.CANBTC.all;

    ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
    ECanaShadow.CANMC.bit.CCR = 0 ;            // Set CCR = 0
    ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;

    ECanaShadow.CANES.all = ECanaRegs.CANES.all;

    do
    {
        ECanaShadow.CANES.all = ECanaRegs.CANES.all;
    } while(ECanaShadow.CANES.bit.CCE != 0 );// Wait for CCE bit to be  cleared

    //
    // Disable all Mailboxes
    //
    ECanaRegs.CANME.all = 0;        // Required before writing the MSGIDs

    EDIS;
}


void MBXwrA(void)
{
    int j;
    volatile struct MBOX *Mailbox  = (void *) 0x6100;
        for(j=0; j<32; j++)
        {
            Mailbox->MSGID.all = 0;
            Mailbox->MSGCTRL.all = 0;
            Mailbox->MDH.all = 0;
            Mailbox->MDL.all = 0;
            Mailbox = Mailbox + 1;
        }
}
//
// InitECanGpio - This function initializes GPIO pins to function as eCAN pins
//
// Each GPIO pin can be configured as a GPIO pin or up to 3 different
// peripheral functional pins. By default all pins come up as GPIO
// inputs after reset.
//
// Caution:
// Only one GPIO pin should be enabled for CANTXA/B operation.
// Only one GPIO pin shoudl be enabled for CANRXA/B operation.
// Comment out other unwanted lines.
//
```

```
void
InitECanGpio(void)
{
    InitECanaGpio();
}

//
// InitECanaGpio - This function initializes GPIO pins to function as eCAN- A
//
void
InitECanaGpio(void)
{
    EALLOW;

    //
    // Enable internal pull-up for the selected CAN pins
    // Pull-ups can be enabled or disabled by the user.
    // This will enable the pullups for the specified pins.
    // Comment out other unwanted lines.
    //
    GpioCtrlRegs.GPAPUD.bit.GPIO30 = 0;   // Enable pull-up for GPIO30 (CANRXA)
    //GpioCtrlRegs.GPAPUD.bit.GPIO18 = 0; // Enable pull-up for GPIO18 (CANRXA)

    GpioCtrlRegs.GPAPUD.bit.GPIO31 = 0;  //Enable pull-up for GPIO31 (CANTXA)
    //GpioCtrlRegs.GPAPUD.bit.GPIO19 = 0;  //Enable pull-up for GPIO19 (CANTXA)

    //
    // Set qualification for selected CAN pins to asynch only
    // Inputs are synchronized to SYSCLKOUT by default.
    // This will select asynch (no qualification) for the selected pins.
    //
    GpioCtrlRegs.GPAQSEL2.bit.GPIO30 = 3;   // Asynch qual for GPIO30 (CANRXA)
    //GpioCtrlRegs.GPAQSEL2.bit.GPIO18 = 3;  // Asynch qual for GPIO18 (CANRXA)

    //
    // Configure eCAN-A pins using GPIO regs
    // This specifies which of the possible GPIO pins will be eCAN functional
    // pins.
    //
    GpioCtrlRegs.GPAMUX2.bit.GPIO30 = 1;    // Configure GPIO30 for CANRXA
    //GpioCtrlRegs.GPAMUX2.bit.GPIO18 = 3;  // Configure GPIO18 for CANRXA
    GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 1;    // Configure GPIO31 for CANTXA
    //GpioCtrlRegs.GPAMUX2.bit.GPIO19 = 3;  // Configure GPIO19 for CANTXA

    EDIS;
}
```

INICIALIZACIÓN DE NUESTRAS VARIABLES

```
//+++INICIALIZACION++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

unsigned int CANTimer ;
unsigned int CANTimertest ;
unsigned int CANTimerACKtimeout;

void CANinit(void){

// Step 1. Initialize System Control:
// PLL, WatchDog, enable Peripheral Clocks
// This example function is found in the DSP280x_SysCtrl.c file.
   InitSysCtrl();
    CANTimertest = 4500;
    CANTimer = 2000;
    CANTimerACKtimeout = 0;
/* Initialize the CAN module */

    InitECan();
    InitECanGpio();
    EALLOW;

/* Zero out (or initialize) the entire MBX RAM area */

    MBXwrA();

/* Write to the MSGID field 0f CAN-A - MBX number is written as its MSGID */

    ECanaMboxes.MBOX4.MSGID.all = 0x00040004;   //RECEIVE

    ECanaMboxes.MBOX10.MSGID.all = 0x000A0004;  //EEPROM

    ECanaMboxes.MBOX11.MSGID.all = 0x000B0004;  //
    ECanaMboxes.MBOX12.MSGID.all = 0x000C0004;  //PROCESSDATA
    ECanaMboxes.MBOX13.MSGID.all = 0x000D0004;  //

    ECanaMboxes.MBOX14.MSGID.all = 0x000E0004;  //WARNINGs AND ERRORS


    ECanaMboxes.MBOX4.MSGID.bit.IDE = 1;        // Extended identifier enable
    ECanaMboxes.MBOX10.MSGID.bit.IDE = 1;
    ECanaMboxes.MBOX11.MSGID.bit.IDE = 1;
    ECanaMboxes.MBOX12.MSGID.bit.IDE = 1;
    ECanaMboxes.MBOX13.MSGID.bit.IDE = 1;
    ECanaMboxes.MBOX14.MSGID.bit.IDE = 1;

    ECanaMboxes.MBOX4.MSGID.bit.AME = 1;        // Acceptance mask enable

    ECanaLAMRegs.LAM4.all = 0b11111111100001111111111111011;    // Add mask

/* Configure CAN-A Mailboxes as Receive/Transmit mailboxes */
```

```
    ECanaShadow.CANMD.all = ECanaRegs.CANMD.all;
    ECanaShadow.CANMD.all = 0xFFFFFFFF;

    ECanaShadow.CANMD.bit.MD10 = 0;              //transmit mailboxes
    ECanaShadow.CANMD.bit.MD11 = 0;
    ECanaShadow.CANMD.bit.MD12 = 0;
    ECanaShadow.CANMD.bit.MD13 = 0;
    ECanaShadow.CANMD.bit.MD14 = 0;
    ECanaRegs.CANMD.all = ECanaShadow.CANMD.all;

/* Enable Mailboxes */

    ECanaShadow.CANME.all = ECanaRegs.CANME.all;
    ECanaShadow.CANME.all = 0xFFFFFFFF;
    ECanaRegs.CANME.all = ECanaShadow.CANME.all;

    ECanaMboxes.MBOX10.MSGCTRL.bit.DLC = 8;     // Data Length Code
    ECanaMboxes.MBOX11.MSGCTRL.bit.DLC = 8;
    ECanaMboxes.MBOX12.MSGCTRL.bit.DLC = 8;
    ECanaMboxes.MBOX13.MSGCTRL.bit.DLC = 8;
    ECanaMboxes.MBOX14.MSGCTRL.bit.DLC = 8;
/*
    Note: If writing only to the 11-bit identifier as by
    "ECanaMboxes.MBOX1.MSGID.bit.STDMSGID = 1;", IDE, AME & AAM bit fields also
     need to be initialized. Otherwise, they may assume random values. This
could
     be done by just initializing the entire register to zero first and then
writing
    the STD MSG ID.
*/
}
//
// Note: Bit timing parameters must be chosen based on the network parameters
// such as the sampling point desired and the propagation delay of the network.
// The propagation delay is a function of length of the cable, delay introduced
// by the transceivers and opto/galvanic-isolators (if any).
//
// The parameters used in this file must be changed taking into account the
// above mentioned factors in order to arrive at the bit-timing parameters
// suitable for a network.
//
```

FUNCIONES

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
int page;
Uint16 CAN_Daten[256] = {0};
```

```
struct MBOX RXMbox;

void sendCANProzessdaten(void){ // Send sensors data
    //MESSAGE 1 -------------------------------------------------------

        ECanaMboxes.MBOX11.MDL.byte.BYTE0 = ADC.Temperatur.OnBoard;
        ECanaMboxes.MBOX11.MDL.byte.BYTE1 = ADC.Temperatur.Magnet;
        ECanaMboxes.MBOX11.MDL.byte.BYTE2 = ADC.Temperatur.KK;
        ECanaMboxes.MBOX11.MDL.byte.BYTE3 = 0;

        ECanaMboxes.MBOX11.MDH.byte.BYTE4 = 0;
        ECanaMboxes.MBOX11.MDH.byte.BYTE5 = 0;
        ECanaMboxes.MBOX11.MDH.byte.BYTE6 = 0;
        ECanaMboxes.MBOX11.MDH.byte.BYTE7 = 0;

    //MESSAGE 2 -------------------------------------------------------

                        ECanaMboxes.MBOX12.MDL.byte.BYTE0      =      ((int)
(ADC.Magnetspannung.floatWert*10)) >> 8;
                        ECanaMboxes.MBOX12.MDL.byte.BYTE1      =      ((int)
(ADC.Magnetspannung.floatWert*10));
                        ECanaMboxes.MBOX12.MDL.byte.BYTE2      =      ((int)
(ADC.Zwischenkreis.floatWert*10)) >> 8;
                        ECanaMboxes.MBOX12.MDL.byte.BYTE3      =      ((int)
(ADC.Zwischenkreis.floatWert*10));

                        ECanaMboxes.MBOX12.MDH.byte.BYTE4      =      ((int)
(ADC.Magnetstrom1.floatWert*100)) >> 8;
                        ECanaMboxes.MBOX12.MDH.byte.BYTE5      =      ((int)
(ADC.Magnetstrom1.floatWert*100));
                        ECanaMboxes.MBOX12.MDH.byte.BYTE6      =      ((int)
(ADC.Magnetstrom2.floatWert*100)) >> 8;
                        ECanaMboxes.MBOX12.MDH.byte.BYTE7      =      ((int)
(ADC.Magnetstrom2.floatWert*100));

    //MESSAGE 3 -------------------------------------------------------

        ECanaMboxes.MBOX13.MDL.byte.BYTE0 = 0;
        ECanaMboxes.MBOX13.MDL.byte.BYTE1 = 0;
        ECanaMboxes.MBOX13.MDL.byte.BYTE2 = 0;
        ECanaMboxes.MBOX13.MDL.byte.BYTE3 = 0;
        ECanaMboxes.MBOX13.MDH.byte.BYTE4 = 0;
        ECanaMboxes.MBOX13.MDH.byte.BYTE5 = 0;
        ECanaMboxes.MBOX13.MDH.byte.BYTE6 = 0;
        ECanaMboxes.MBOX13.MDH.byte.BYTE7 = 0;

    // --Transmit-------------------------------------------------------
        ECanaShadow.CANTRS.all = 0;
        ECanaShadow.CANTRS.bit.TRS11 = 1;
        ECanaShadow.CANTRS.bit.TRS12 = 1;
```

```
        ECanaShadow.CANTRS.bit.TRS13 = 1;
        ECanaRegs.CANTRS.all = ECanaShadow.CANTRS.all;
        CANTimerACKtimeout = 100;
            do {
                ECanaShadow.CANTA.all = ECanaRegs.CANTA.all;
                if(CANTimerACKtimeout == 0) break;
                }   // WAIT ACKNOWLEDGE
            while((ECanaShadow.CANTA.bit.TA13 && ECanaShadow.CANTA.bit.TA12 &&
ECanaShadow.CANTA.bit.TA11) == 0);

        ECanaShadow.CANTA.all = 0;              // RESET REGISTER
        ECanaShadow.CANTA.bit.TA11 = 1;         // Clear TA
        ECanaShadow.CANTA.bit.TA12 = 1;
        ECanaShadow.CANTA.bit.TA13 = 1;
        ECanaRegs.CANTA.all = ECanaShadow.CANTA.all;
}


void read_EEPromCAN(struct strEEProm *EEProm)  // Read EEPROM memory and load
it in CAN_Daten
{
    tempInitEE = I2C_Read(EEPROM_ADRESSE,EEProm->Start_Adresse,256,CAN_Daten);
    if(tempInitEE != I2C_OK)
        return EEPROM_ERROR;
    return EEPROM_OK;
}

void sendCANWarnungFehler(void){    // Send Warnings and error from event list
    if((Warning.all != 0) || (Error.all != 0) || nuevaAccion){
        CANread_EEPromEreignisliste();
        int p=0;
        int i=0;
        for(i=0; i<32; i++){
            p=i*8;
            ECanaMboxes.MBOX14.MDL.byte.BYTE0 = CAN_Daten[p];
            ECanaMboxes.MBOX14.MDL.byte.BYTE1 = CAN_Daten[p+1];
            ECanaMboxes.MBOX14.MDL.byte.BYTE2 = CAN_Daten[p+2];
            ECanaMboxes.MBOX14.MDL.byte.BYTE3 = CAN_Daten[p+3];

            ECanaMboxes.MBOX14.MDH.byte.BYTE4 = CAN_Daten[p+4];
            ECanaMboxes.MBOX14.MDH.byte.BYTE5 = CAN_Daten[p+5];
            ECanaMboxes.MBOX14.MDH.byte.BYTE6 = CAN_Daten[p+6];
            ECanaMboxes.MBOX14.MDH.byte.BYTE7 = CAN_Daten[p+7];

            ECanaShadow.CANTRS.all = 0;
            ECanaShadow.CANTRS.bit.TRS14 = 1;
            ECanaRegs.CANTRS.all = ECanaShadow.CANTRS.all;
            CANTimerACKtimeout = 100;
                do {
                    ECanaShadow.CANTA.all = ECanaRegs.CANTA.all;
```

```
                    if(CANTimerACKtimeout == 0) break;
                }   // WAIT ACKNOWLEDGE
                while(ECanaShadow.CANTA.bit.TA14 == 0);

            ECanaShadow.CANTA.all = 0;                // RESET REGISTER
            ECanaShadow.CANTA.bit.TA14 = 1;
            ECanaRegs.CANTA.all = ECanaShadow.CANTA.all;
        }
    }
}

void sendCANDatenEEprom(int page){  // Send EEPROM data
    int p=0;
    int i=0;
    for(i=0; i<32; i++){
        p=i*8;
        ECanaMboxes.MBOX10.MDL.byte.BYTE0 = CAN_Daten[p];
        ECanaMboxes.MBOX10.MDL.byte.BYTE1 = CAN_Daten[p+1];
        ECanaMboxes.MBOX10.MDL.byte.BYTE2 = CAN_Daten[p+2];
        ECanaMboxes.MBOX10.MDL.byte.BYTE3 = CAN_Daten[p+3];
        ECanaMboxes.MBOX10.MDH.byte.BYTE4 = CAN_Daten[p+4];
        ECanaMboxes.MBOX10.MDH.byte.BYTE5 = CAN_Daten[p+5];
        ECanaMboxes.MBOX10.MDH.byte.BYTE6 = CAN_Daten[p+6];
        ECanaMboxes.MBOX10.MDH.byte.BYTE7 = CAN_Daten[p+7];

        // Transmit starts
        ECanaShadow.CANTRS.all = 0;
        ECanaShadow.CANTRS.bit.TRS10 = 1;
        ECanaRegs.CANTRS.all = ECanaShadow.CANTRS.all;
        CANTimerACKtimeout = 100;
        do {
            ECanaShadow.CANTA.all = ECanaRegs.CANTA.all;
            if(CANTimerACKtimeout == 0) break;
        }   // WAIT ACKNOWLEDGE
        while(ECanaShadow.CANTA.bit.TA10 == 0);

        ECanaShadow.CANTA.all = 0;                // RESET REGISTER
        ECanaShadow.CANTA.bit.TA10 = 1;
        ECanaRegs.CANTA.all = ECanaShadow.CANTA.all;
    }

    sendChecksum(page);
}

void sendChecksum(int page){
    unsigned int checksum=0;
    int i=0;
    CANTimertest = 20;
    for( i=0; i<256; i++){
        checksum += CAN_Daten[i];   // Checksum calculation
```

```
    }
    // MSG 33
    ECanaMboxes.MBOX10.MDL.byte.BYTE0 = checksum;
    ECanaMboxes.MBOX10.MDL.byte.BYTE1 = checksum >> 8;
    ECanaMboxes.MBOX10.MDL.byte.BYTE2 = 0;
    ECanaMboxes.MBOX10.MDL.byte.BYTE3 = 0;
    ECanaMboxes.MBOX10.MDH.byte.BYTE4 = 0;
    ECanaMboxes.MBOX10.MDH.byte.BYTE5 = 0;
    ECanaMboxes.MBOX10.MDH.byte.BYTE6 = 0;
    ECanaMboxes.MBOX10.MDH.byte.BYTE7 = page;

        // Transmit
        ECanaShadow.CANTRS.all = 0;
        ECanaShadow.CANTRS.bit.TRS10 = 1;
        ECanaRegs.CANTRS.all = ECanaShadow.CANTRS.all;
        CANTimerACKtimeout =100;
        do {
            ECanaShadow.CANTA.all = ECanaRegs.CANTA.all;
            if(CANTimerACKtimeout == 0) break;
            }   // WAIT ACKNOWLEDGE
        while(ECanaShadow.CANTA.bit.TA10 == 0);

        ECanaShadow.CANTA.all = 0;                  // RESET REGISTER
        ECanaShadow.CANTA.bit.TA10 = 1;
        ECanaRegs.CANTA.all = ECanaShadow.CANTA.all;
    while(CANTimertest > 0){} // Wait until the counter get 0
}


void receiveCAN(void){
    if(ECanaRegs.CANRMP.bit.RMP4 == 1){ // Received message?
        // Compare what values has changed from last message
        if(RXMbox.MDL.byte.BYTE0 != ECanaMboxes.MBOX4.MDL.byte.BYTE0){
                                Steuerung.Anfrage    =    (STEUERUNG_ANFRAGE)
ECanaMboxes.MBOX4.MDL.byte.BYTE0;
        }
        if(RXMbox.MDL.byte.BYTE1 != ECanaMboxes.MBOX4.MDL.byte.BYTE1){
            if((ECanaMboxes.MBOX4.MDL.byte.BYTE1 <= 3)){
                Eingaenge.OptionenIN5_8 = ECanaMboxes.MBOX4.MDL.byte.BYTE1;
            }
        }
        if(ECanaMboxes.MBOX4.MDL.byte.BYTE2 <= 2){
            read_EEPromCAN(EEProm.Page1);
            sendCANDatenEEprom(1);
            read_EEPromCAN(EEProm.Page2);
            sendCANDatenEEprom(2);
            read_EEPromCAN(EEProm.Page3);
            sendCANDatenEEprom(3);
            read_EEPromCAN(EEProm.Page4);
            sendCANDatenEEprom(4);
```

```
            read_EEPromCAN(EEProm.Page5);
            sendCANDatenEEprom(5);
            read_EEPromCAN(EEProm.Page6);
            sendCANDatenEEprom(6);
        }
        if(RXMbox.MDL.byte.BYTE3 != ECanaMboxes.MBOX4.MDL.byte.BYTE3){
           Magnet1.Magnettyp = (MAGNET_TYP) ECanaMboxes.MBOX4.MDL.byte.BYTE3;
        }
        if(RXMbox.MDH.byte.BYTE4 != ECanaMboxes.MBOX4.MDH.byte.BYTE4){

        }
        if(RXMbox.MDH.byte.BYTE5 != ECanaMboxes.MBOX4.MDH.byte.BYTE5){

        }
        if(RXMbox.MDH.byte.BYTE6 != ECanaMboxes.MBOX4.MDH.byte.BYTE6){

        }
        if(ECanaMboxes.MBOX4.MDH.byte.BYTE7 <= 2){
            //...
            write_EEPROM();
            //...
        }

        ECanaRegs.CANRMP.bit.RMP4 = 1;  // Reset register
        RXMbox = ECanaMboxes.MBOX4;     // Save message
    }
}
```

FUNCIÓN MAIN

```
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

void main(void){
    Inicializar_Sistema();
    CANinit();
    // Comenzar bucle Loop()
    for(;;){
        receiveCAN();
        if(CANTimer == 0){
            sendCANProzessdaten();
            sendCANWarnungFehler();
            CANTimer = 1000;
        }

        if(Error.all != prevError || Warning.all != PrevWarn){
            sendCANWarnungFehler();
        }

        //.....
```

```
        // Mas rutinas
        //.....
    }
}
```

7.2 ESP32

DEFINICIONES E INICIALIZACIÓN DE VARIABLES

```
#include "Arduino.h"
#include "clsPCA9555.h"
#include <TFT_eSPI.h>
#include <ESP32-TWAI-CAN.hpp>

// Port definitions for the CAN bus
#define CAN_TX 13
#define CAN_RX 12
// Port definitions for the display
#define TFT_CS     2
#define TFT_DC     19
#define TFT_MOSI   23
#define TFT_SCLK   18
#define TFT_RST    15

PCA9555 ioport(0x20,);    // Port expander I2C address and interrupt pin

CanFrame rxFrame;

int valueEncoder1 = 0;
int valueEncoder2 = 0;
// Variables de la EEPROM
typedef struct {
    uint8_t Daten[256];
    uint8_t checksum;
} strEEprom;
typedef struct {
    uint16_t ID;
    strEEprom EEprom[PAGE_MAX];
} strDevice;

strEEprom EEprom[PAGE_MAX];
int page;
uint8_t CAN_Daten[256];
int checksum;
int EEpromByte=0;
bool WarnungFehlerReceivedFlag = false;

int numDevices = 1;
```

```
strDevice Device[1];
// Variables de sensores
float TemperaturOnBoard;
float TemperaturMagnet;
float TemperaturKK;
float Magnetspannung = 0;
float Zwischenkreis;
float Magnetstrom1;
float Magnetstrom2;
// Variables de control
int steuerungAnfrage = 0;
int magnetAuswahl = 3;
int EEpromReadFlag = 0;
int EEpromWriteFlag = 0;
int Magnettyp = 0;

// Variables del display

#define GFXFF 17 // Stock font and GFXFF reference handle
#define CUSTOMFONT &FreeSans9pt7b // Easily remembered name for the font

TFT_eSPI tft = TFT_eSPI();

int refreshDisplayFlag = false;

const char* menuItems[] = {"Instruccion", "Puerto iman", "Leer EEprom",
"Escribir EEprom", "Tipo Iman", "Datos sensores", "Eventos", "Item 8"};
const int menuLength = sizeof(menuItems) / sizeof(menuItems[0]);
int currentMenuIndex = 0;
bool inMenu = true;
bool inSubMenu = false;
int currentSubMenuIndex = -1;
int MenuSelectedIndex = 0;

bool selectButton = false;
bool backButton = false;

int EreignisStrColor[8];
char EreignisStr[8][30];

const char* subMenuItems[][10] = {
    {"Ninguna", "Magnetizar", "Zurueckregeln", "Abtippen",
"Desmagnetizar", "Calibrar", "KONFIG", "ADC_NULL", "Quit", "OFF"},
    {"MAGNETWAHL", "UMPOLPROGRAMM", "HAFTKRAFT", "KEINE_OPTIONEN_IN5_8"},
    {"Enviar"},
    {"Escribir"},
    {"Emagnet", "EPmagnet", "Ndmagnet"},
```

```
    {nullptr , nullptr },
    {nullptr}
};
const int subMenuLengths[] = {
    10, // Number of options in subMenuItems[i]
    4,
    2,
    2,
    3,
    2
    // Add more lengths as needed
};

char EE_Fehlertext[32][30] = {
  "E00:\tAC-Supply out\t",
  "E01:\tDC-Link out\t",
    // ....
};
char EE_Warntext[32][30] = {
  "W00:\tEarth fault\t",
  "W01:\tOB Temperature\t",
  //......
};

char magnetstrom1Str[22];
char magnetstrom2Str[22];
```

FUNCIONES:

```
//=====================================================================//
void setup() {
    Serial.begin (115200);
    while (!Serial);
    delay (1000);

    // It is also safe to use .begin() without .end() as it calls it
internally
    if (ESP32Can.begin(ESP32Can.convertSpeed(1000), CAN_TX, CAN_RX, 198,
198)) {
        Serial.println("CAN bus started!");
    } else {
        Serial.println("CAN bus failed!");
    }

    // start I2C
    ioport.begin();
```

```
    ioport.setClock(400000);    // sets the I2C clock to 400kHz
    for (uint8_t i = 0; i < 13; i++){
        ioport.pinMode(i, INPUT);   // config port expander pins
    }
    // start Display
    tft.begin();
    tft.fillScreen(TFT_BLACK);
    tft.setTextWrap(false);
    // initialize device struct
    initDevices();
}
//=====================================================================//

void loop() {

    static uint32_t lastStamp    = 0;
    uint32_t        currentStamp = micros();

    if (ESP32Can.readFrame(rxFrame, 0)) {   // Si se recibe un mensaje
CAN, clasificar el mensaje
        sortMessage(0);
    }

    if(((currentStamp - lastStamp) > 10000) ||
WarnungFehlerReceivedFlag){  // Refrescar la pantalla del display cada
10ms o si se recibe la lista de eventos
        readEncoder();
        handleButtonPresses();
        if (inMenu || inSubMenu) {
            handleMenuNavigation();
        }
    }
}
```

FUNCIONES CAN

```
//=== FUNCIONES CAN ===================================================//

void canSendMessage(uint32_t Id = 0x00040004) {
    Serial.print ("Sending packet 2 ... ");

    CanFrame txFrame         = {0};
    txFrame.identifier       = Id;
    txFrame.extd             = 1;
    txFrame.data_length_code = 8;

    txFrame.data[0]          = steuerungAnfrage;
```

```
    txFrame.data[1]          = magnetAuswahl;
    txFrame.data[2]          = EEpromReadFlag;
    txFrame.data[3]          = Magnettyp; // Best to use 0xAA
(0b10101010) instead of 0,
    txFrame.data[4]          = 0xAA; // CAN works better this way as it
needs
    txFrame.data[5]          = 0xAA; // to avoid bit-stuffing
    txFrame.data[6]          = 0xAA;
    txFrame.data[7]          = EEpromWriteFlag;

    if(ESP32Can.writeFrame(txFrame)){  // timeout defaults to 1 ms
        Serial.println("done");
    }
    else{
        Serial.println("failed");
    }
    EEpromReadFlag = 100;
    EEpromWriteFlag = 100;
}

void sortMessage(int packetSiz) {
    // Leer a partir del bit 16 del ID
    if((rxFrame.identifier >> 16) == 0x000A){
      receiveEEprom();
    }
    else if(((rxFrame.identifier >> 16) == 0x000B) ||
((rxFrame.identifier >> 16) == 0x000C) || ((rxFrame.identifier >> 16) ==
0x000D)){
      receiveSensorsData(rxFrame.identifier);
    }
    else if((rxFrame.identifier >> 16) == 0x000E){
        receiveWarnungFehler();
    }
}
int checksuma(){
    checksum = 0;
    int checksumReceived = 0;

    for(int j = 0; j < 256; j++){
        checksum += CAN_Daten[j];
    }

    // Assign checksum and page of the message n.33
    checksumReceived += rxFrame.data[0];
    checksumReceived += (rxFrame.data[1] << 8);
    page = rxFrame.data[7];
```

```
    for(int i=0;i<8;i++){   // Print Checksum and page
        Serial.print(rxFrame.data[i],HEX);
        Serial.print(" ");
    }
    Serial.println();

    if(checksum == checksumReceived){
        Serial.println("Checksum correcto");
        if(page > PAGE_MAX || page < 0){
            Serial.println("Pagina incorrecta");
            return 0;
        }
        return page;
    }
    else{
        Serial.println("Checksum incorrecto");
        Serial.print("Checksum calculado: ");
        Serial.println(checksum);
        Serial.print("Checksum recibido: ");
        Serial.println(checksumReceived);
        return 0;
    }
}


void receiveSensorsData(uint32_t id){
    if(id == 0x000B0004){
        TemperaturOnBoard = (int8_t) rxFrame.data[0];
        TemperaturMagnet = (int8_t) rxFrame.data[1];
        TemperaturKK = (int8_t) rxFrame.data[2];
    }
    if(id == 0x000C0004){
        Magnetspannung = (int16_t) (rxFrame.data[0] << 8);
        Magnetspannung = (int16_t) Magnetspannung | (int16_t)
rxFrame.data[1];
        Magnetspannung /= 10.0;
        Zwischenkreis = (int16_t) (rxFrame.data[2] << 8);
        Zwischenkreis = (int16_t) Zwischenkreis | (int16_t)
rxFrame.data[3];
        Zwischenkreis /= 10.0;
        Magnetstrom1 = (int16_t) (rxFrame.data[4] << 8);
        Magnetstrom1 = (int16_t) Magnetstrom1 | (int16_t)
rxFrame.data[5];
        Magnetstrom1 /= 100.0;
        Magnetstrom2 = (int16_t) (rxFrame.data[6] << 8);
        Magnetstrom2 = (int16_t) Magnetstrom2 | (int16_t)
rxFrame.data[7];
        Magnetstrom2 /= 100.0;
```

```
        snprintf(magnetstrom1Str, 22, "Voltaje: %.1f V", Magnetspannung);
        snprintf(magnetstrom2Str, 22, "Intens.: %.2f I", Magnetstrom2);
        // Assign the converted strings to the subMenuItems array
        subMenuItems[5][0] = magnetstrom1Str;
        subMenuItems[5][1] = magnetstrom2Str;

        refreshDisplayFlag = true;
    }
    if(id == 0x000D0004){

    }
}

void receiveEEprom(void){
    int ptr=0;
    int sum = 0;
    EEprom_page_t EepromPage;

    if(EepromByte < 32){
        for(int i=0;i<8;i++){
            CAN_Daten[i+EepromByte*8] = rxFrame.data[i];
        }
        EepromByte++;
    }
    else{
        EepromPage = (Eeprom_page_t) checksuma();

        if(EepromPage == 0){
            EepromByte=0;
            return;       // If the checksum is incorrect, not save the
data
        }
        Eeprom[EepromPage].checksum = checksum;
        for(int i=0;i<256;i++){
            Eeprom[EepromPage].Daten[i] = CAN_Daten[i];
        }
        EepromByte=0;
    }
}

void receiveWarnungFehler(void){     // Recibir lista de eventos
    static int BytePtr = 0;
    for(int i=0; i<8; i++){
        Eeprom[PAGE_6].Daten[i+BytePtr*8] = rxFrame.data[i];

    }
```

```
    BytePtr++;
    if(BytePtr == 32 || BytePtr < 0){
        WarnungFehlerReceivedFlag = true;
        BytePtr = 0;
    }
}

void initDevices(void) {

    for(int I = 0; I < numDevices; i++){
        Device[i].ID = 1 << I;
        for(int j = 0; j < PAGE_MAX; j++){
            for(int k = 0; k < 256; k++){
                Device[i].Eeprom[j].Daten[k] = 0;
            }
            Device[i].Eeprom[j].checksum = 0;
        }
    }
    Serial.print("Device size: ");
    Serial.println(sizeof(strDevice)*numDevices);
}
```

FUNCIONES DEL MENU Y OTRAS FUNCIONES

```
//=== FUNCIONES DEL MENU Y OTRAS ====================================//

void displayMenu() {
    tft.setFreeFont(CUSTOMFONT);
    tft.fillScreen(TFT_BLACK);
    tft.setCursor(0, 30);
    tft.setTextColor(TFT_WHITE, TFT_BLACK);
    tft.setTextSize(1);

    int startIndex = currentMenuIndex - 2;
    if (startIndex < 0) {
        startIndex += menuLength;
    }

    for (int i = 0; (i < 5); i++) {
        int index = (startIndex + i) % menuLength;
        if (index == currentMenuIndex) {
            tft.setTextColor(TFT_RED, TFT_BLACK); // Highlight selected
item
        } else {
            tft.setTextColor(TFT_WHITE, TFT_BLACK);
        }
        tft.println(menuItems[index]);
```

```
    }
}

void displaySubMenu(int title, int currentMenuIndex) {
    tft.fillScreen(TFT_BLACK);
    tft.setCursor(15, 20);
    tft.setTextColor(TFT_RED, TFT_BLACK);
    tft.setTextSize(1);
    tft.println(menuItems[title]);
    tft.setTextColor(TFT_WHITE, TFT_BLACK);

    int startIndex = currentMenuIndex - 2;
    if (startIndex < 0) {

        startIndex = 0;
    }

    for (int i = 0; (i < 5); i++) {
        int index = (startIndex + i);
        if (index == (currentMenuIndex % subMenuLengths[title])) {
            tft.setTextColor(TFT_RED, TFT_BLACK); // Highlight selected
item
        } else {
            tft.setTextColor(TFT_WHITE, TFT_BLACK);
        }
        if(index < subMenuLengths[title]) {
            tft.println(subMenuItems[title][index]);
        }
        else {
            tft.println(" ");
        }
    }
}

void handleMenuNavigation() {     // Gestiona la navegacion del menu
    static int lastMenuIndex = -1;

    if (valueEncoder1 >= 0) {
        currentMenuIndex = valueEncoder1 % (inSubMenu ? 16 : menuLength);
    } else {
        currentMenuIndex = (valueEncoder1 + (inSubMenu ? 16 :
menuLength)) % (inSubMenu ? 16 : menuLength);
    }
    if (currentMenuIndex != lastMenuIndex) {
        if (inSubMenu) {
            if(MenuSelectedIndex == 6){
                decodificarEreignis();
```

```
                displayEreignis();
            } else if(MenuSelectedIndex == 5){
                displaySubMenu(MenuSelectedIndex, currentMenuIndex);
            } else {
                displaySubMenu(MenuSelectedIndex, currentMenuIndex);
            }
        }
        else {
            displayMenu();
        }
        lastMenuIndex = currentMenuIndex;
    }

    if(inSubMenu && (MenuSelectedIndex == 5) && refreshDisplayFlag){
        displaySubMenu(MenuSelectedIndex, currentMenuIndex);
        refreshDisplayFlag = false;
    }
    if(inSubMenu && (MenuSelectedIndex == 6) &&
WarnungFehlerReceivedFlag){
        decodificarEreignis();
        displayEreignis();
        WarnungFehlerReceivedFlag = false;
    }
}

void handleButtonPresses() {        // Gestiona la accion de los botones
    bool newSelectButtonState = !numRead(8);
    bool newBackButtonState = !numRead(10);
    static bool oldSelectButtonState;
    static bool oldBackButtonState;

    if (newSelectButtonState != oldSelectButtonState) {
        selectButton = newSelectButtonState;    // Select button state
changed
    }

    if (newBackButtonState != oldBackButtonState) {
        backButton = newBackButtonState;        // Back button state
changed
    }

    if (selectButton) { // Select button
        if (inSubMenu) {
            switch (MenuSelectedIndex) {
                case 0:
                    steuerungAnfrage = valueEncoder1;
                    printf("steuerungAnfrage: %d\n", steuerungAnfrage);
```

```
                break;
            case 1:
                magnetAuswahl = valueEncoder1;
                printf("Magnetauswahl: %d\n", magnetAuswahl);
                break;
            case 2:
                EEpromReadFlag = valueEncoder1;
                printf("EEpromRead: %d\n", EEpromReadFlag);
                break;
            case 3:
                EEpromWriteFlag = valueEncoder1;
                printf("EEpromWrite: %d\n", EEpromWriteFlag);
                break;
            case 4:
                Magnettyp = valueEncoder1;
                printf("Magnettyp: %d\n", Magnettyp);
                break;
            default:
                break;
        }
        canSendMessage();
        inSubMenu = true;
        inMenu = false;
    } else {
        inMenu = false;
        inSubMenu = true;
        MenuSelectedIndex = currentMenuIndex;
        if(MenuSelectedIndex == 6){
            decodificarEreignis();
            displayEreignis();
        }
        else{
            displaySubMenu(MenuSelectedIndex, currentMenuIndex);
        }
    }
    selectButton = false;

} else if (backButton) { // Back button
    if (inSubMenu) {
        inSubMenu = false;
        inMenu = true;
        displayMenu();
    } else {
        inMenu = true;
        displayMenu();
    }
}
```

```
    oldSelectButtonState = newSelectButtonState;
    oldBackButtonState = newBackButtonState;
    selectButton = false;
    backButton = false;
}

void readEncoder(){
    valueEncoder2 = (int) encoderRead(2);
    valueEncoder1 = (int) encoderRead(1);
}

uint8_t encoderRead(int n){
    uint8_t dat = 0;
    int i = n*4 -1;
    int end = i-4;
    if(n == 0){
        i = 3;
    }
    for ( i; i >= end ; i--){
        dat |= ioport.digitalRead(i) << (i-end-1);
    }
    return dat;
}

uint8_t numRead(int n){
    return ioport.digitalRead(n);
}

bool isEreignisStrInitialized() {
    for (int i = 0; i < 8; i++) {
        if (strlen(EreignisStr[i]) > 0) {
            return true;
        }
    }
    return false;
}

void displayEreignis(void){     //  Mostrar lista de eventos en la
pantalla
    tft.fillScreen(TFT_BLACK);
    tft.setCursor(0, 30);
    tft.setTextColor(TFT_WHITE, TFT_BLACK);
    tft.setTextSize(1);

    tft.println("Eventos: ");
    tft.setTextFont(1);
```

```
    if(isEreignisStrInitialized()){
        for (int i = 0; i < 8; i++) {
            if(EreignisStrColor[i] == 2){
                tft.setTextColor(TFT_RED, TFT_BLACK);
            } else if(EreignisStrColor[i] == 1){
                tft.setTextColor(TFT_YELLOW, TFT_BLACK);
            } else if(EreignisStrColor[i] == 0){
                tft.setTextColor(TFT_GREEN, TFT_BLACK);
            }

            tft.println(EreignisStr[i]);
        }
    }
    else{
        tft.println("No hay eventos");
    }
    tft.setFreeFont(CUSTOMFONT);
}

void decodificarEreignis(void){     //  Decodificar la lista de eventos
recibida por CAN

    // Vaciar EreignisStr[i] y EreignisStrColor[i],
    // Decodificar lista de eventos y añadir eventos decodificados a
EreignisStr[i] y su color a EreignisStrColor[i] mediante bucle for

}
```