Trabajo Fin de Grado

Integración de dispositivos serie en buses CAN

Autor: *Eleazar Pérez Arencibia*

Tutores: *Antonio Carlos Domínguez Brito*

Jorge Cabrera Gámez

RESUMEN

Esta propuesta de Trabajo de Fin de Grado se basa en la integración de dispositivos con interfaz serie, normalmente UART, en un bus CAN mediante el diseño, programación y desarrollo de un dispositivo que haga de interfaz serie a bus CAN de forma transparente.

Este dispositivo tendrá la capacidad de obtener datos de dispositivos serie e integrarlos en un bus CAN. Dichos datos estarán además arbitrados mediante el protocolo de ventana deslizante que se ha implementado para mejorar el flujo de datos a través de la red CAN.

ABSTRACT

This proposal of End-of-Grade Work is based on the integration of devices with serial interface, usually UART, on a CAN bus by designing, programming and developing a device that makes serial interface to CAN bus in a transparent way.

This device will have the ability to obtain data from serial devices and integrate them into a CAN bus.Those data will further be arbitrated by the sliding window protocol that has been implemented to upgrade the data flow through the CAN network.

Índice de Contenido

RESUMEN 2

ABSTRACT 4

1. Introducción 11

1.1 Estado actual 11

1.2 Objetivos Iniciales 11

1.3 Justificación de las competencias específicas encubiertas 12

2. Material utilizado 15

2.1 Hardware 15

2.2 Software 15

3. Descripción de los dispositivos utilizados 17

3.1 Arduino UNO 17

3.2 SPARKFUN CAN-BUS SHIELD 19

3.3 Dispositivos Serie 20

4. Desarrollo del proyecto 22

4.1 Bus CAN 22

4.1.1 Introducción al bus CAN 22

4.1.2 Tipos de bus CAN 22

4.1.3 Capa física del bus CAN 23

4.1.3.1 Niveles de tensión 24

4.1.3.2 Cables y conectores 25

4.1.3.3 Sincronización y arbitraje 26

4.1.4 Capa de enlace de datos del bus CAN 27

4.1.4.1 Acceso al medio 28
4.1.4.2 Tipos de tramas 28
4.1.4.2.1 Trama de datos 28
4.1.4.2.2 Trama remota 33
4.1.4.2.3 Trama de error 33
4.1.4.2.4 Trama de sobrecarga 34
4.1.4.3 Separación entre tramas 34
4.1.4.4 Bit stuffing 35
4.1.5 Comunicación entre nodos 35
4.1.6 Detección de errores 36
4.1.7 MCP2515 38
4.1.7.1 Características Básicas 38
4.1.7.2 Bloque de protocolo SPI 42
4.1.7.3 Modos de funcionamiento 42
4.1.7.4 Recepción de mensajes 46
4.1.7.4.1 Buffers de recepción de mensajes 46
4.1.7.4.2 Prioridad en la recepción de mensajes 47
4.1.7.4.3 Filtros de aceptación de mensajes 48
4.1.7.5 Transmisión de mensajes 49
4.1.7.5.1 Buffers de transmisión de mensajes 49
4.1.7.5.2 Prioridad en la transmisión de mensajes 49
4.1.7.5.3 Iniciar la transmisión de mensajes 50
4.1.7.5.4 Abortar la transmisión de mensajes 51

4.1.7.6 Temporización y sincronización de bits 51

4.1.7.6 .1 Temporización 51

4.1.7.6.2 Sincronización.. 54

4.1.7.6.3 Programación de los segmentos de tiempo y configuración de los registros 55

4.1.8 Librería ArCan 56

4.2 Comunicación Serie 64

4.2.1 Librería Serial 65

4.3 Protocolo de ventana deslizante.. 69

4.3.1 Funcionamiento ventana de transmisión................................ 69

4.3.2 Funcionamiento ventana de recepción.................................... 71

4.3.3 Recuperación de errores 72

4.4 Integración UART-CAN-Ventana deslizante............................. 74

4.4.1 Librería Timer 75

4.4.2 Librería SlidWind.. 76

4.5 Pruebas realizadas 85

5. Conclusiones.. 92

6. Trabajos Futuros 95

7. Agradecimientos 98

8.Referencias 100

9.Anexo.. 103

9.1. Anexo 1 103

9.2. Anexo 2 104

Tabla de Ilustraciones

Ilustración 1: Arduino UNO.Fuente [10] 17

Ilustración 2: CAN-BUS Shield de Sparkfun. Fuente [11] 19

Ilustración 3: Distribución CAN-BUS Shield de Sparkfun. Fuente: [12] 20

Ilustración 4: Terminación del bus CAN de alta velocidad. Fuente: [4] 23

Ilustración 5: Terminación del bus CAN de baja velocidad tolerante a fallos. Fuente: [4] . 23

Ilustración 6: Niveles de tensión del bus CAN. Fuente: [3] 24

Ilustración 7: Conector D-sub 9 pines CANBUS. Fuente: [3] 25

Ilustración 9: Características físicas de los cables del bus. Fuente: [4] 26

Ilustración 10: Ejemplo de cuantificación de un bit CAN con 10 cuantos de tiempo por bit. Fuente: [3] 27

Ilustración 11: Formato de la trama de datos estándar. Fuente: [5] 29

Ilustración 12: Formato de la trama de datos extendida. Fuente: [5] 31

Ilustración 13: Formato de la trama remota con identificador extendido. Fuente: [5] 33

Ilustración 14: Formato de la trama de error. Fuente: [5] 34

Ilustración 15: Formato de la trama de sobrecarga 34

Ilustración 16: Ejemplo de uso de máscaras y filtros.Fuente [3] 35

Ilustración 17: Evolución entre estados de error. Fuente: [3] 38

Ilustración 18: Esquema MCP2515. Fuente: [5] 40

Ilustración 19: Esquema MCP2551. Fuente: [5] 40

Ilustración 20: Conexión entre MCP2515 y MCP2551 Fuente: [5] y [6] 41

Ilustración 21: Bloque lógico general con el bloque SPI. Fuente: [5] 42

Ilustración 22: Registro CANCTRL. Fuente: [5] 45

Ilustración 23: Registro CANSTAT. Fuente: [5] 45

Ilustración 24: Registro CANINTF. Fuente: [5] 45

Ilustración 25: Registro CANINTE. Fuente: [5] 45

Ilustración 26: Tabla resumen de los registros de control. Fuente: [5] 46

Ilustración 27: Diagrama de bloques de los buffers de recepción. Fuente: [5] 47

Ilustración 28: Ecuación tasa de bit nominal. Fuente: [5] 52

Ilustración 29: Segmentos de tiempo de bit CAN. Fuente: [5] 52

Ilustración 30: Ecuación tiempo de bit nominal. Fuente: [5] 52

Ilustración 31: Ecuación para calcular el Quantum de Tiempo(TQ). Fuente: [5] 54

Ilustración 32: Registro librería ArCan 57

Ilustración 33: Comunicación entre dos dispositivos UART. Fuente: [14] 65

Ilustración 34: Funcionamiento ventana deslizante emisor con temporizadores. Fuente: Wikipedia 71

Ilustración 35: Funcionamiento ventana deslizante con recuperación de errores. Fuente: Wikipedia 73

Ilustración 36: Esquema integración UART-CAN-Ventana_deslizante 74

Ilustración 37: Registro entry 76

Ilustración 38: Registro sender 77

Ilustración 39: Registro recipient 78

Ilustración 40: Registro timeout 79

Ilustración 41: Esquema UART-CAN-VENTANA_DESLIZANTE para la realización de pruebas 85

Ilustración 42: Gráfica tiempos de transmisión 87

Ilustración 43: Gráfica tiempos de transferencia 87

Ilustración 45: Esquema integración UART-CAN-Ventana_deslizante con nodo centralizador 95

1. Introducción

1.1 Estado actual

El denominado bus CAN (Controller Area Network) permite la integración de múltiples dispositivos empotrados basados en microcontroladores en un bus compartido robusto frente a ruido y con anchos de banda considerables para este tipo de dispositivos. Por otro lado, se trata de un bus serie conformado desde un punto de vista hardware por un par de cables entrelazados que unen físicamente a los dispositivos involucrados, que normalmente tienen una taxonomía variada yendo desde los pequeños dispositivos sensores y/o actuadores conectados al bus, dispositivos con mayor potencia computacional, así como otros dispositivos que pueden hacer de puente o pasarela entre dicho bus y otros mecanismos de comunicación (ethernet, radio, wifi, etc.).

En general, existen en el mercado numerosos dispositivos con distintas funcionalidades (sensores, actuadores, etc.) que presentan una interfaz hardware serie (conexión RS232) como medio de acceso y control por parte de otro dispositivo supervisor.

1.2 Objetivos Iniciales

Como objetivo principal de esta propuesta de TFG se ha planteado la integración de dispositivos con interfaz serie, normalmente UART (RS232, etc.), en un bus CAN mediante el diseño y desarrollo de un dispositivo que haga de interfaz serie a bus CAN de forma transparente.

Para llevar a cabo este tipo de interfaz se usaron varios Arduino Uno [10] con el CAN-BUS Shield de Sparkfun[11], de manera que el Arduino Uno[10] será el encargado de recibir la información del monitor serial que posee el propio dispositivo y el CAN-BUS Shield el encargado de enviar la información por nuestro bus CAN hacia otro dispositivo.

Otro objetivo es la minimización del uso de pines del Arduino Uno para llevar a cabo este tipo de comunicaciones, ya que, para comunicarnos con un dispositivo serie por medio del Arduino es necesario el uso de 2 pines, de manera que, con esa propuesta en cada nodo solo usaremos 2 pines para comunicaciones serie y 4 pines, ya especificados en el CAN-BUS Shield de Sparkfun, para la comunicación CAN, dejando el resto de pines libres para cualquier otra utilidad.

Como último objetivo es la integración del concepto de red en los microcontroladores, aprovechando el bus CAN, centralizando la información en un solo nodo de esta red "Serie-CANBUS".

1.3 Justificación de las competencias específicas encubiertas

A continuación, se exponen las competencias específicas cubiertas en este Trabajo de Fin de Grado con su respectiva justificación:

Competencia IC01 → Capacidad de diseñar y construir sistemas digitales, incluyendo computadores, sistemas basados en microprocesador y sistemas de comunicaciones.

Se ha conectado entre si distintos componentes: el Arduino Uno con el CANBUS Shield de Sparkfun, y a su vez todo esto conectado a una red que usa protocolo de comunicación CAN.

Competencia IC02 → Capacidad de desarrollar procesadores específicos y sistemas empotrados, así como desarrollar y optimizar el software de dichos sistemas.

Se ha desarrollado software para: la comunicación a través del bus CAN, la comunicación con el monitor serie del propio Arduino UNO [10], y se desarrolló un mecanismo de ventana deslizante para un control y mejora del flujo de datos.

Competencia IC04 → Capacidad de diseñar e implementar software de sistema y de comunicaciones.

Desarrollo de software para las comunicaciones tanto CAN bus como serie.

Competencia IC05 → Capacidad de analizar, evaluar y seleccionar las plataformas hardware y software más adecuadas para el soporte de aplicaciones empotradas y de tiempo real.

Selección del dispositivo CANBUS Shield de Sparkfun que dispone del microcontrolador MCP2515 y para su desarrollo software de las comunicaciones CANBUS se partió de la librería ArCan, añadiéndole funcionalidades y mejoras.

Competencia IC07 → Capacidad para analizar, evaluar, seleccionar y configurar plataformas hardware para el desarrollo y ejecución de aplicaciones y servicios informáticos.

Configuración del bus CAN de manera que las comunicaciones serán a través de tramas de tipo extendido y remotas, aunque la red soportará el resto de tipos disponibles en el protocolo CAN.

Implementación en el protocolo de ventana deslizante de tres tipos de paquetes para un mejor control y mejora en el flujo de datos.

2. Material utilizado

El material utilizado en este Trabajo Fin de Grado se puede dividir en dos secciones: hardware y software.

2.1 Hardware

Se ha utilizado el siguiente hardware:

- Tarjeta de prototipado Arduino UNO.
- CANBUS Shield de Sparkfun.
- Componentes electrónicos y eléctricos (Cables de conexión, 2 resistencias de 120 Ω).

2.2 Software

El software usado fue el siguiente:

- Entorno de desarrollo Arduino IDE [13].
- Librería para el manejo del bus CAN "ArCan" [12].
- Librería para el manejo del monitor serial del Arduino UNO [15].

3. Descripción de los dispositivos utilizados

3.1 Arduino UNO

Arduino UNO se trata de una placa de prototipado electrónico basada en el microcontrolador ATmega328P.

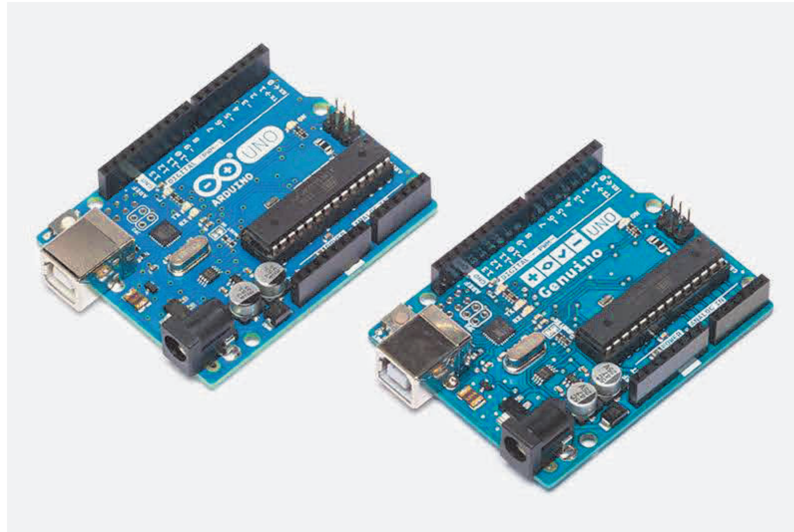


Ilustración 1: Arduino UNO.Fuente [10]

Cuenta con 14 pines digitales de entrada / salida (de los cuales 6 se pueden utilizar como salidas PWM), 6 entradas analógicas, un cristal de cuarzo de 16 MHz, una conexión USB, un conector de alimentación, una cabecera ICSP y un botón de reinicio. Contiene todo lo necesario para apoyar el microcontrolador; basta con conectarlo a un ordenador con un cable USB o la corriente con un adaptador de CA a CC.

Sus especificaciones técnicas son las siguientes:

Microcontrolador	**ATmega328P**
Tensión de funcionamiento	5V
Voltaje de entrada (recomendado)	7-12V

Voltaje de entrada (límite)	6-20V
Digital pines I / O	14 (de los cuales 6 proporcionan una salida PWM)
PWM digital pines I / O	6
Pines de entrada analógica	6
Corriente DC por Pin I / O	20 mA
Corriente CC para Pin 3.3V	50 mA
Memoria flash	32 KB (ATmega328P) de los cuales 0,5 KB utilizado por cargador de arranque
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Velocidad de reloj	16 MHz
Longitud	68,6 mm
Anchura	53,4 mm
Peso	25 g

La placa Arduino UNO nos permite aprovechar muchas de sus cualidades hardware, mediante las cuales por medio de software nos permite añadirle distintas funcionalidades.

3.2 SPARKFUN CAN-BUS SHIELD

Este shield creado por la empresa Sparkfun nos permite dar la capacidad de manejar el bus CAN con el Arduino Uno

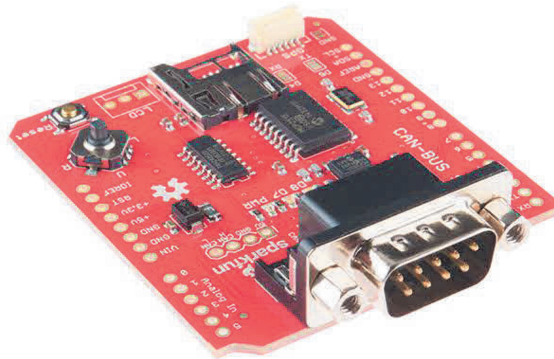
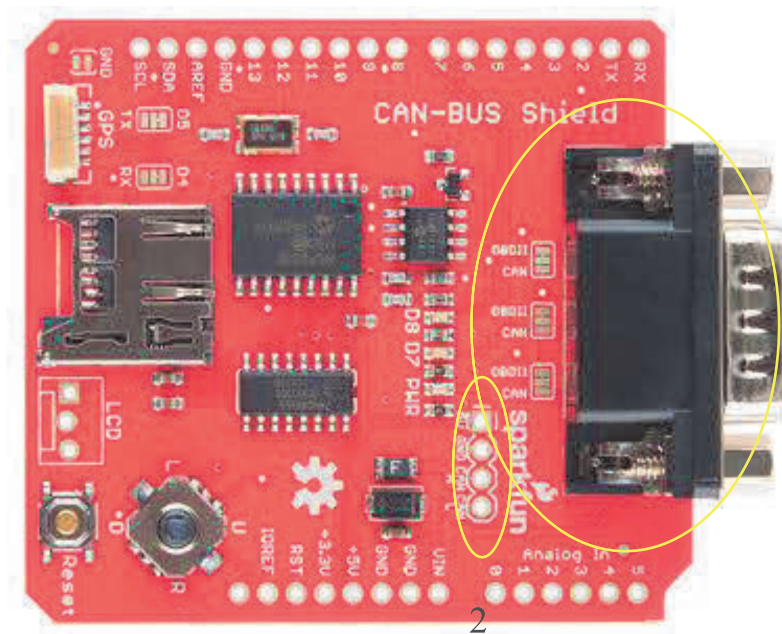


Ilustración 2: CAN-BUS Shield de Sparkfun. Fuente [11]

Este shield usa el microcontrolador MCP2515 con el transceptor MCP2551.Las conexiones al bus CAN se pueden realizar de dos maneras: a través de un conector de 9 vías sub-D (1) para su uso mediante el cable OBD-II, y mediante la conexión directa al bus mediante el conector de 4 vías (2) que son 5 voltios, CAN H, CAN L, GND. Ambas conexiones están mapeadas en la siguiente ilustración:



1

Ilustración 3: Distribución CAN-BUS Shield de Sparkfun. Fuente: [12]

Este shield también dispone de un módulo para tarjeta SD, conector para un LCD serie, un conector para GPS EM506, y un control de joystick. A continuación, presentamos algunas de sus características técnicas más importantes:

- CAN v2.0B hasta 1 Mb/s.
- Interfaz de alta velocidad SPI (10 MHz).
- Tramas estándar, extendidas y remotas.
- Puede suministrar electricidad al Arduino por el conector sub-D a través de fusibles reajustables y protección de polaridad inversa.
- Botón de reset.
- Dos indicadores LED.

3.3 Dispositivos Serie

Los dispositivos usados para las comunicaciones serie fueron los mismos Arduino UNO a través del monitor serie, ya introducidos en el apartado 3.1

4. Desarrollo del proyecto

En los siguientes apartados se hará una breve descripción de la tecnología en cuestión y luego se realizará una explicación sobre su implementación de cara al desarrollo de este TFG.

4.1 Bus CAN

4.1.1 Introducción al bus CAN

El bus CAN fue desarrollado inicialmente para aplicaciones en los automóviles y por lo tanto la plataforma del protocolo es resultado de las necesidades existentes en el área de automoción. Se basa en el modelo productor/consumidor, el cual es un concepto, o paradigma de comunicaciones de datos, que describe una relación entre un productor y uno o más consumidores. Es un protocolo orientado a mensajes, es decir la información que se va a intercambiar se descompone en mensajes, a los cuales se les asigna un identificador de mensaje y se encapsulan en tramas para su transmisión. Cada mensaje tiene su identificador que es único dentro de la red, con el cual, los nodos deciden aceptar o no dicho mensaje.

En general, está definido para proveer de una comunicación determinista en sistemas distribuidos complejos. Nos proporciona prioridad de mensajes y garantiza latencias máximas, así como una sincronización orientada a bit y una consistencia de datos de todo el sistema. También tiene la capacidad de dar acceso al bus a varios masters (multimaster), capacidad de detección y corrección de errores, retransmisión automática y detectar si el dispositivo está desconectado del bus, así como capacidad de aislamiento de nodos.

4.1.2 Tipos de bus CAN

La especificación de los buses CAN está recogida en el conjunto de estándares ISO 11898 [15]. Dicha especificación define las dos primeras capas, la capa física y la capa de enlace de datos, del modelo OSI de interconexión de sistemas. Sobre la base de dichos estándares, los buses CAN se pueden clasificar en dos tipos:

CAN de alta velocidad (hasta 1 Mbit/s) → También conocido como ISO11898-2, usa un único bus lineal terminado en cada extremo con resistencias de 120Ω, tal y como está especificada la impedancia característica del bus, para evitar reflexiones en la línea que podrían perturbar la comunicación.

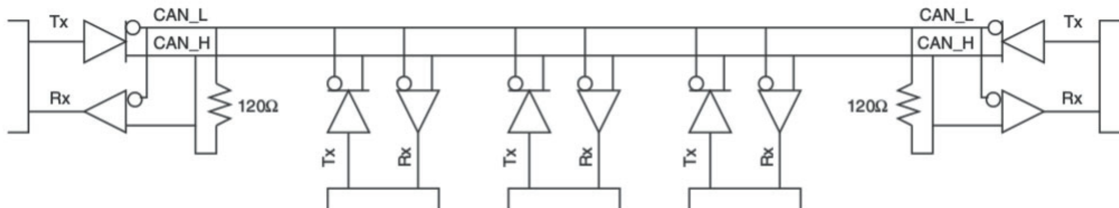


Ilustración 4: Terminación del bus CAN de alta velocidad. Fuente: [4]

CAN de baja velocidad tolerante a fallos (hasta 125 kbit/s) → También conocido como ISO11898-3, puede utilizar un bus lineal, un bus en estrella o múltiples buses en estrella conectados por un bus lineal. Independiente de su topología el bus está terminado en cada nodo por una fracción de la resistencia de terminación total, la cual tiene un valor de 9KΩ.

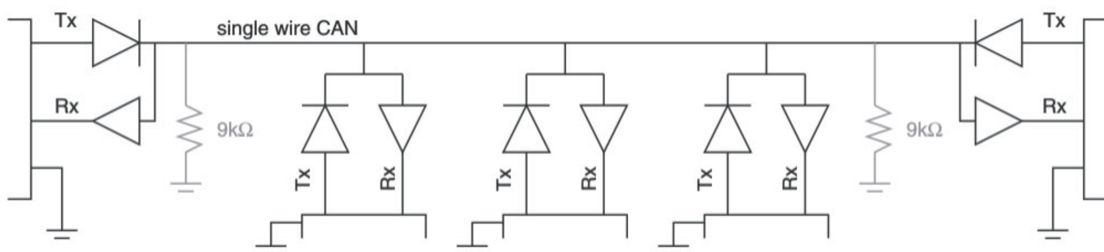


Ilustración 5: Terminación del bus CAN de baja velocidad tolerante a fallos. Fuente: [4]

4.1.3 Capa física del bus CAN

En la capa física se encuentran los aspectos del medio físico que son necesarios para la transmisión de datos entre nodos de una red CAN, las características materiales y eléctricas y la transmisión del flujo de bits a través del bus.

4.1.3.1 Niveles de tensión

La transmisión de señales en un bus CAN se lleva a cabo a través de dos cables trenzados. Las señales de estos cables se denominan CAN_H (CAN high) y CAN_L (CAN low) respectivamente. El bus tiene dos estados definidos: estado dominante y estado recesivo. En estado recesivo, los dos cables del bus se encuentran al mismo nivel de tensión, mientras que en estado dominante hay una diferencia de tensión entre CAN_H y CAN_L de al menos 1,5 V. La transmisión de señales en forma de tensión diferencial, en comparación con la transmisión en forma de tensiones absolutas, proporciona protección frente a interferencias electromagnéticas.

La tensión en modo común puede estar, según la especificación, en cualquier punto entre -2 y 7 V. La tensión diferencial del bus (la diferencia entre CAN_H y CAN_L) en modo dominante debe estar entre 1,5 y 3 V. Esto permite la conexión directa entre nodos que operen a distintas tensiones, e incluso nodos que sufran diferencia de tensión entre sus respectivas tierras.

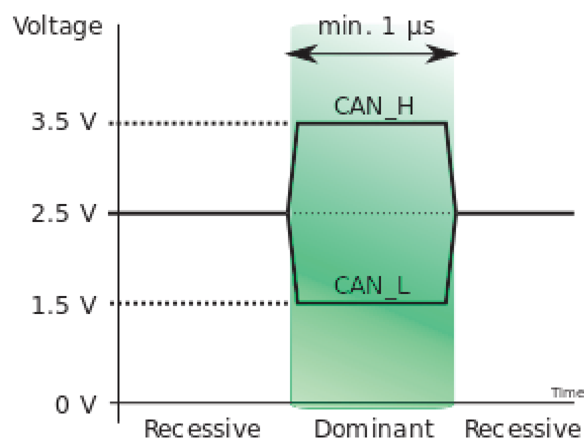


Ilustración 6: Niveles de tensión del bus CAN. Fuente: [3]

4.1.3.2 Cables y conectores

Los distintos nodos de un bus CAN deben estar interconectados mediante un par de cables trenzados con una impedancia característica de 120 Ω, y puede ser cable apantallado o sin apantallar. El cable trenzado proporciona protección frente a interferencias electromagnéticas externas. Y si, además, está apantallado, la protección será mayor, pero a cambio de un incremento en el coste del cable.

El estándar CAN, a diferencia de otros estándares como el USB, no especifica ningún tipo de conector para el bus y por lo tanto cada aplicación puede tener un conector distinto. Sin embargo, hay varios formatos comúnmente aceptados como el conector D-sub de 9 pines, con la señal CAN_L en el pin 2 y la señal CAN_H en el pin 7.

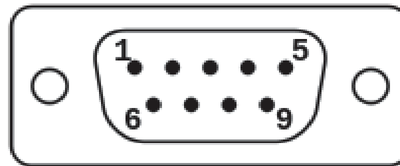


Ilustración 7: Conector D-sub 9 pines CANBUS. Fuente: [3]

Las propiedades de la línea de transmisión limitan el ancho de banda de los datos. Orientativamente, se aceptan los siguientes valores como tasa de transferencia y retardo de propagación en función de la longitud del bus:

Bit rate	Bit time (μs)	Bus length (m)
1 Mb/s	1	30
800 kb/s	1.25	50
500 kb/s	2	100
250 kb/s	4	250
125 kb/s	8	500
62.5 kb/s	16	1,000
20 kb/s	50	2,500
10 kb/s	100	5,000

Ilustración 8: Relación de tasas de transferencia, retardo de propagación y longitud del bus. Fuente: [4]

También es conveniente tener en cuenta las características de los cables del bus especificadas a continuación:

Bus speed	Cable type	Cable resistance/m	Terminator	Bus length
50 kb/s at 1,000 m	0.75–0.8 mm^2 (AWG18)	70 m Ω	150–300 Ω	600–1,000 m
100 kb/s at 500 m	0.5–0.6 mm^2 (AWG20)	<60 m Ω	150–300 Ω	300–600 m
500 kb/s at 100 m	0.34–0.6 mm^2 (AWG22, AWG20)	<40 m Ω	127 Ω	40–300 m
1,000 kb/s at 40 m	0.25–0.34 mm^2 (AWG23, AWG22)	<26 mΩ	124 Ω	0–40 m

Ilustración 9: Características físicas de los cables del bus. Fuente: [4]

4.1.3.3 Sincronización y arbitraje

Todos los nodos de un bus CAN deben trabajar con la misma tasa de transferencia nominal. Dado que el bus CAN no usa una señal de reloj separada, factores como el retardo de la señal de reloj y la tolerancia de los osciladores causan que haya una diferencia entre la tasa de transferencia real de los distintos nodos. Por ello es necesario un método de sincronización entre los nodos. Esta sincronización es especialmente importante en la fase de arbitraje ya que, durante esta fase, cada nodo debe ser capaz de observar tanto los datos transmitidos por él como los datos transmitidos por los demás nodos.

El requisito mínimo para un bus CAN es que dos nodos, estando en los extremos opuestos de la red con el máximo retardo de propagación entre ellos, y cuyos controladores CAN tienen unas frecuencias de reloj en los límites opuestos de la tolerancia de frecuencia especificada, sean capaces de recibir y leer correctamente todos los mensajes transmitidos por la línea. Esto incluye que todos los nodos muestreen el valor correcto de cada bit.

Cada controlador CAN conectado al bus espera que se produzca una transición del bus de recesivo a dominante en un determinado intervalo de tiempo. Si dicha transición no ocurre en el intervalo esperado, el controlador reajusta la duración del siguiente bit en consecuencia. Dicho ajuste se lleva a cabo dividiendo cada bit en intervalos o cuantos de tiempo y asignando los intervalos a los cuatro segmentos de cada bit:

- Segmento de sincronización (Sync): Es el intervalo de tiempo en el que se supone que ocurren las transiciones de recesivo a dominante.
- Segmento de propagación (Prop): Es el intervalo de tiempo que compensa los retardos de propagación a lo largo de la línea.

- Segmentos de fase 1 y 2 (Phase 1 y Phase 2): Se usan para llevar a cabo la re-sincronización de los nodos. El segmento de fase 1 puede ser alargado o el 2 acortado para dicha re-sincronización. El punto de muestreo del bit se encuentra inmediatamente después del segmento de fase 1. El punto de muestreo se encuentra habitualmente cerca del 75 % de la duración total del bit.

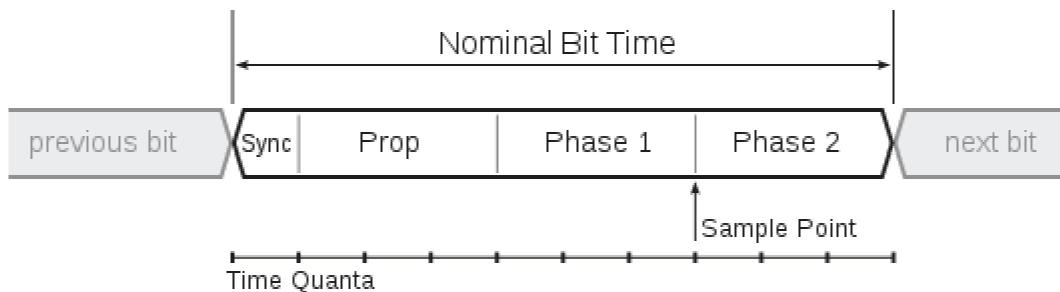


Ilustración 10: Ejemplo de cuantificación de un bit CAN con 10 cuantos de tiempo por bit. Fuente: [3]

La configuración de los segmentos del bit se hace sobre la base de la frecuencia de reloj de cada controlador CAN. Los segmentos se configuran individualmente para cada controlador en un mismo bus. A efectos prácticos, la configuración de los segmentos del bit supone un compromiso entre la tasa de transferencia y tolerancia de los osciladores.

4.1.4 Capa de enlace de datos del bus CAN

El protocolo CAN proporciona un acceso multimaestro al bus con una resolución determinista de las colisiones. La capa de enlace de datos define el método de acceso al medio, así como los tipos de tramas para el envío de mensajes.

4.1.4.1 Acceso al medio

La especificación del CAN usa los términos “dominante” y “recesivo” para referirse a los bits, donde un bit dominante equivale al valor lógico 0 y un bit recesivo equivale al valor lógico 1. El estado inactivo del bus es el estado recesivo (valor lógico 1). Cuando dos nodos intentan transmitir bits diferentes se denomina colisión y el valor del bit dominante prevalece sobre el valor del bit recesivo. En ese caso el nodo que intentaba transmitir el valor recesivo detecta la colisión y pasa a modo pasivo, es decir, deja de transmitir para escuchar lo que transmite el otro nodo. Por esta razón es importante que todos los nodos estén sincronizados y muestreen todos los bits del bus simultáneamente.

El arbitraje se produce durante los primeros bits de una trama o mensaje, durante la transmisión de lo que se conoce como identificador del mensaje. Al final del proceso de arbitraje sólo debe quedar un nodo con el control del bus. Por ello cada nodo debe manejar identificadores únicos. Cuando un nodo pierde el arbitraje aplaza la transmisión de su trama para intentarlo de nuevo cuando finalice la trama actual. Conociendo los identificadores de todas las tramas que intentan ser transmitidas, se puede establecer de manera determinista el orden en el que son transmitidas. Así, una trama CAN con identificador más bajo (mayor número de bits dominantes en las primeras posiciones) tiene más prioridad que una trama con identificador más alto.

4.1.4.2 Tipos de tramas

Existen cuatro tipos de tramas CAN: Tramas de datos (data frame), tramas remotas (remote frame), tramas de error (error frame), tramas de sobrecarga (overload frame).

4.1.4.2.1 Trama de datos

Una trama de datos CAN puede ser de uno de los dos siguientes formatos:

- Formato estándar: con identificador de 11 bits.
- Formato extendido: con identificador de 29 bits.

La especificación del bus dice que un controlador CAN debe aceptar tramas en formato estándar, y puede o no aceptar tramas en

formato extendido. Pero en cualquier caso debe tolerar tramas en formato extendido. Es decir, que si un controlador está configurado para que sólo acepte tramas en formato estándar no debe lanzar un error cuando reciba una trama en formato extendido, sino que simplemente no transmitirá el mensaje al procesador central.

El formato de la trama estándar es el siguiente:

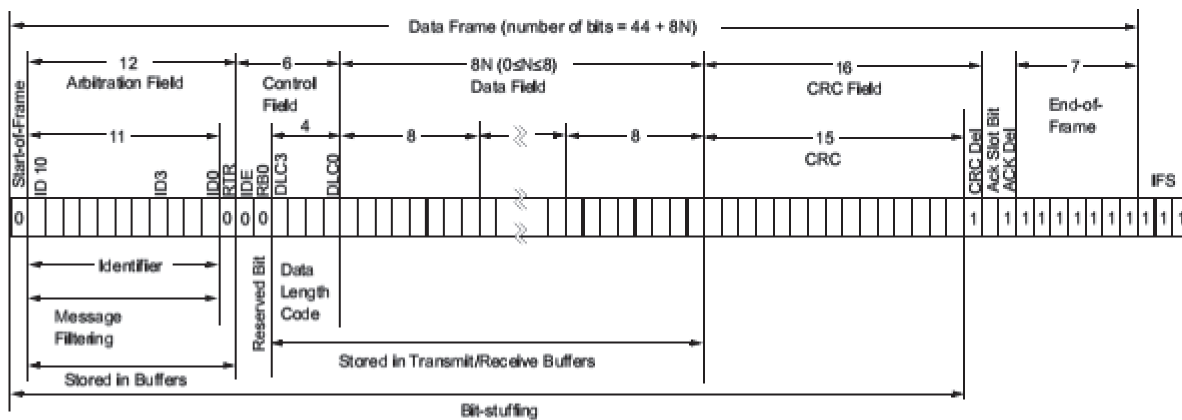


Ilustración 11: Formato de la trama de datos estándar. Fuente: [5]

Nombre del campo	**Longitud (bits)**	**Explicación**
Start-Of-Frame	1	Indica 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 identificador estándar (identificador de 11 bits).
Bit reservado (RB0)	1	Bit reservado. Debe ser dominante (0), pero aceptado tanto dominante como recesivo.
Código de longitud de datos - DLC	4	Indica el 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.
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 (CRC Del)	1	Debe ser recesivo (1).
Acuse de recibo-ACK Slot Bit	1	El transmisor emite recesivo (1) y cualquier receptor emite dominante (0).
Delimitador ACK- ACK Del	1	Debe ser recesivo (1).

Fin de trama End-Of-Frame	7	Deben ser 7 bits recesivos seguidos (1).

El formato extendido de la trama de datos es el siguiente:

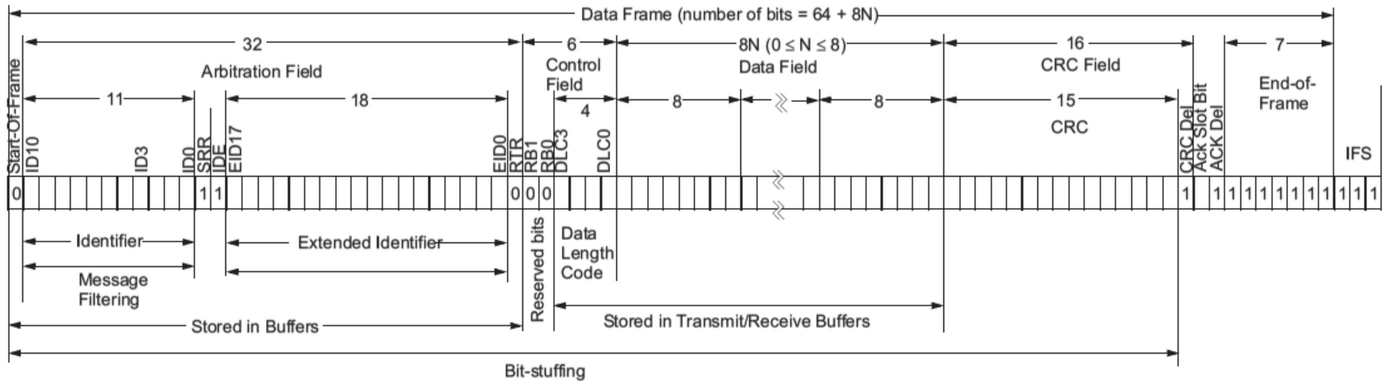


Ilustración 12: Formato de la trama de datos extendida. Fuente: [5]

Nombre del campo	**Longitud (bits)**	**Uso**
Start-Of-Frame	1	Demarca el comienzo de una transmisión.
Identificador - ID	11	Primera parte del identificador (único) que también representa la prioridad de la trama.
Sustituto de transmisión remota - SRR	1	Debe ser recesivo (1).
Bit de extensión de identificador - IDE	1	Recesivo (1) para el formato extendido (identificador de 29 bits).

Identificador Extendido - EID	18	Segunda parte del 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.
Bits reservados (RB1, RB0)	2	Bits reservados. 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).
Acuse de recibo – ACK Slot Bit	1	El transmisor emite recesivo (1) y cualquier receptor emite dominante (0).

Delimitador ACK-ACK Del	1	Debe ser recesivo (1).
Fin de trama - EOF	7	Deben ser 7 bits recesivos seguidos (1).

4.1.4.2.2 Trama remota

Es posible que un nodo requiera unos datos desde otro nodo. En ese caso, el primero puede enviar una trama remota para pedir el envío de algún dato. Este nodo envía entonces una trama con una petición de transmisión remota (RTR = 1; recesivo). Este tipo de tramas sólo se diferencian de las tramas de datos en que las tramas remotas no tienen campo de datos.

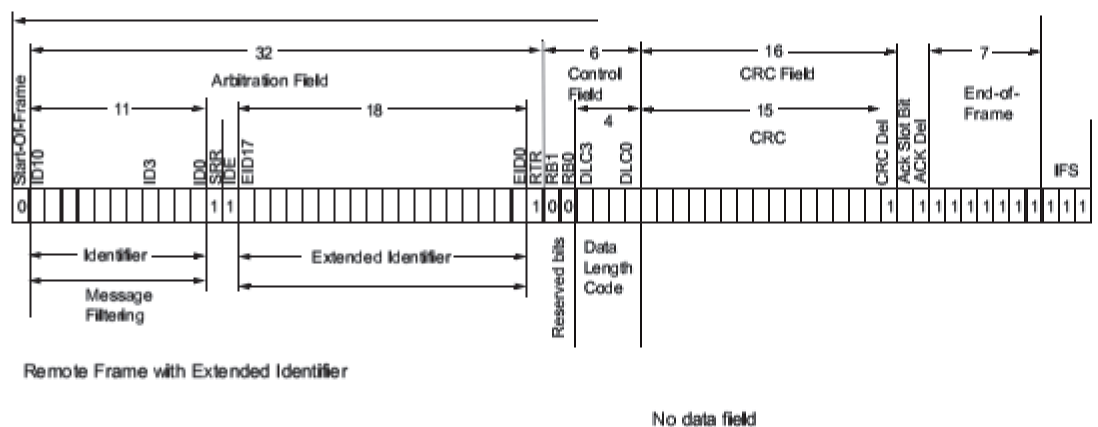


Ilustración 13: Formato de la trama remota con identificador extendido. Fuente: [5]

4.1.4.2.3 Trama de error

La trama de error es una trama especial que viola las reglas, ya que, posee más de 7 bits en el mismo nivel lógico, incumpliendo el formato de las tramas CAN. Se transmite cuando un nodo detecta un mensaje erróneo, y provoca que los demás nodos también transmitan una trama de error. Un complejo mecanismo de contadores de error integrado en el controlador asegura que un nodo no bloquee el bus con continuas tramas de error.

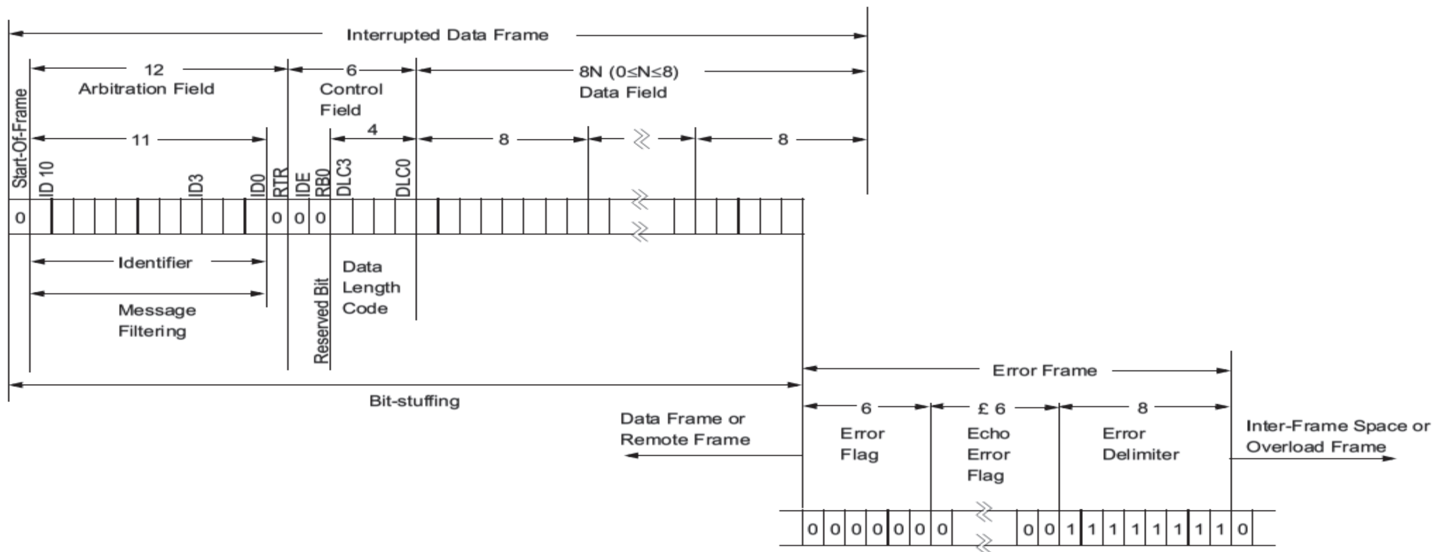


Ilustración 14: Formato de la trama de error. Fuente: [5]

4.1.4.2.4 Trama de sobrecarga

Es similar a la trama de error en cuanto a que viola el formato de las tramas CAN, produciéndose la misma situación que en las tramas de error. Es transmitida por un nodo que se encuentra muy ocupado y el bus proporciona entonces un retardo extra entre tramas.

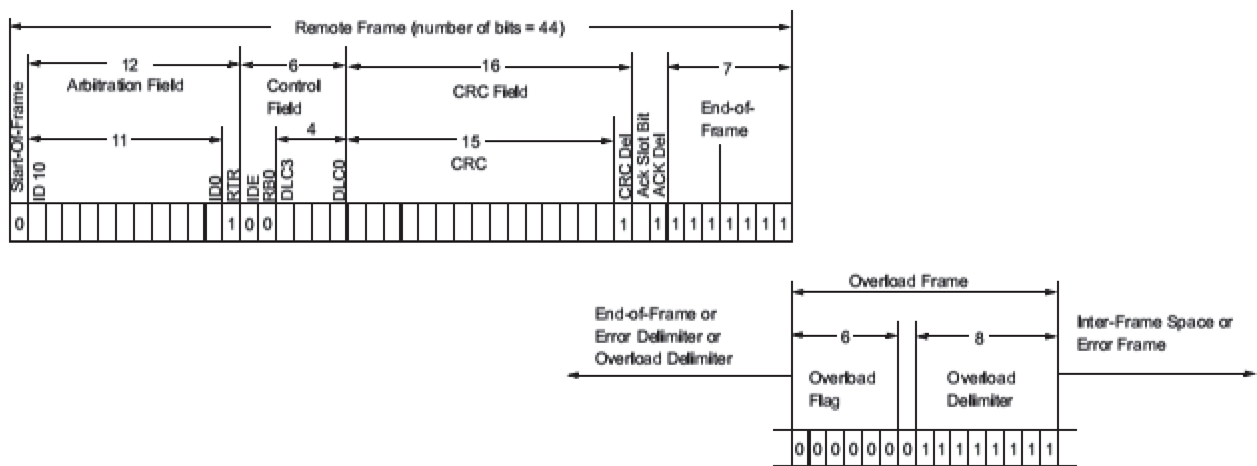


Ilustración 15: Formato de la trama de sobrecarga

4.1.4.3 Separación entre tramas

Las tramas de datos y remotas están separadas por al menos tres bits recesivos (1). Después de eso, si se detecta un bit dominante (0), es

considerado como el inicio de una nueva trama. Las tramas de error y de sobrecarga no respetan el espaciado entre tramas.

4.1.4.4 Bit stuffing

Para asegurar que hay suficientes transiciones recesivo-dominante y garantizar así la sincronización, un bit de nivel lógico opuesto es insertado después de cinco bits consecutivos del mismo nivel lógico. Estos bits insertados son eliminados por el receptor.

Todos los campos de la trama son rellenados a excepción del delimitador CRC, el acuse de recibo ACK, y el fin de trama. Cuando un nodo detecta seis bits consecutivos iguales en un campo susceptible de ser rellenado lo considera un error y emite un error activo. Un error activo consiste en seis bits consecutivos dominantes y viola la regla de relleno de bits (Trama de error activa).

La regla de los bits de relleno implica que una trama puede ser más larga de lo esperado si se suman a estos, los bits de cada campo de la trama.

4.1.5 Comunicación entre nodos

La comunicación entre los nodos se produce por medio de máscaras y filtros, de forma que, la máscara nos va a indicar los bits del identificador de la trama recibida que tenemos que comparar con el filtro establecido, para una mejor comprensión del funcionamiento, establecemos el siguiente ejemplo:

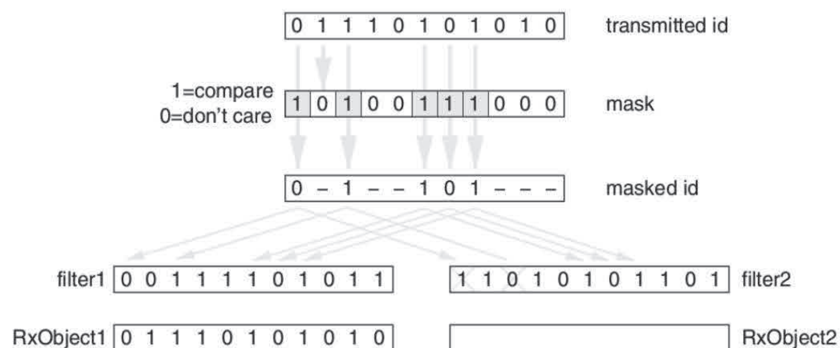


Ilustración 16: Ejemplo de uso de máscaras y filtros.Fuente [3]

El nodo conectado al bus observa que se está produciendo un envío de un paquete, obtiene el id del mismo (transmitted id) que tiene una longitud de 11 bits. Su máscara que también ha de ser de la misma longitud del id, se establece de forma que si el bit de máscara está a 1 se aplica en el filtro y si el bit de máscara está a cero, entonces este bit será aceptado automáticamente independientemente del bit de filtro (masked id). Una vez aplicada esta máscara al id de la trama comprobamos que los bits obtenidos sean iguales al filtro, si esto es así quiere decir que la trama es para el nodo y la recibo, si no es así el nodo hace caso omiso a la transmisión de la misma.

Dependiendo si las tramas usan formato estándar o extendido (bit ide=1), las máscaras y los filtros serán de distinto tamaño.

4.1.6 Detección de errores

Una de las características más importantes y útiles de CAN es su alta fiabilidad, incluso en entornos de ruido extremos, el protocolo CAN proporciona una gran variedad de mecanismos para detectar errores en las tramas. Esta detección de error es utilizada para retransmitir la trama hasta que sea recibida con éxito. Otro tipo de error es el de aislamiento, y se produce cuando un dispositivo no funciona correctamente y un alto por ciento de sus tramas son erróneas. Este error de aislamiento impide que el mal funcionamiento de un dispositivo condicione el funcionamiento del resto de nodos implicados en la red.

Detección de errores

En el momento en que un dispositivo detecta un error en una trama, este dispositivo transmite una secuencia especial de bits (error flag o trama de error activa). Cuando el dispositivo que ha transmitido la trama errónea detecta el error flag, transmite la trama de nuevo. Los dispositivos CAN detectan los errores siguientes:

- **Error de bit:** Durante la transmisión de una trama, el nodo que transmite monitoriza el bus. Cualquier bit que reciba con polaridad inversa a la que ha transmitido se considera un error de bit, excepto cuando se recibe durante el campo de arbitraje(arbitraje de bus, colisión en el envío) o en el bit de reconocimiento. Además, no se

considera error de bit la detección de bit dominante por un nodo en estado de error pasivo que retransmite una trama de error pasivo (trama de error con 7 bits recesivos seguidos).

- **Error de relleno (Stuff Error):** Se considera error de relleno la detección de 6 bits consecutivos del mismo signo, en cualquier campo que siga la técnica de relleno de bits, donde por cada 5 bits iguales se añade uno diferente.
- **Error de CRC:** Se produce cuando el cálculo de CRC realizado por un receptor no coincide con el recibido en la trama. El campo CRC (Cyclic Redundant Code) contiene 15 bits y una distancia de Hamming de 6, lo que asegura la detección de 5 bits erróneos por mensaje. Estas medidas sirven para detectar errores de transmisión debido a posibles incidencias en el medio físico como por ejemplo el ruido.
- **Error de forma (Form Error):** Se produce cuando un campo de formato fijo se recibe alterado como bit.
- **Error de reconocimiento (Acknowledgement Error):** Se produce cuando ningún nodo cambia a dominante el bit de reconocimiento. Si un nodo advierte alguno de los fallos mencionados, iniciará la transmisión de una trama de error. El protocolo CAN especifica diversos fallos en la línea física de comunicación, como por ejemplo línea desconectada, problemas con la terminación de los cables o líneas cortocircuitadas. Sin embargo, no especificará cómo reaccionar en caso de que se produzca alguno de estos errores.

Aislamiento de nodos

Para evitar que un nodo en problemas condicione el funcionamiento del resto de la red, se han incorporado a la especificación CAN medidas de aislamiento de nodos defectuosos que son gestionadas por los controladores. Un nodo puede encontrarse en uno de los tres estados siguientes en relación a la gestión de errores:

- **Error Activo (Error Active):** Es el estado normal de un nodo. Participa en la comunicación y en caso de detección de error envía una trama de error activa.
- **Error pasivo (Error Passive):** Un nodo en estado de error pasivo participa en la comunicación, sin embargo, ha de esperar una secuencia adicional de bits recesivos antes de transmitir y sólo puede señalar errores con una trama de error pasivo.
- **Anulado (Bus Off):** En este estado, deshabilitará su transceptor y no participará en la comunicación. La evolución entre estos estados se basa en dos contadores incluidos en el controlador de comunicaciones. Contador de errores de transmisión (TEC) y Contador de errores de recepción (REC).

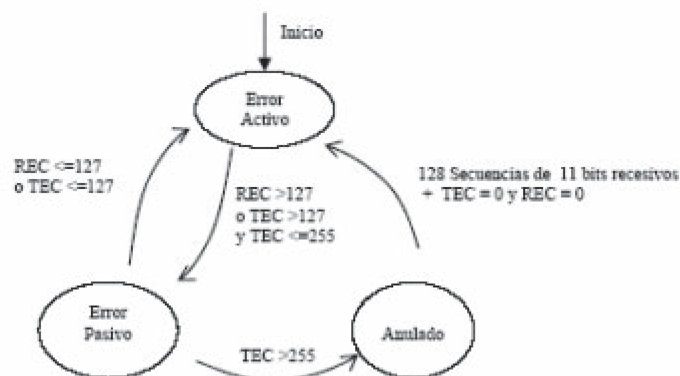


Ilustración 17: Evolución entre estados de error. Fuente: [3]

4.1.7 MCP2515

4.1.7.1 Características Básicas

El shield utilizado para el desarrollo de este T.F.G. tiene integrado el microcontrolador MCP2515[5], que es un sistema autónomo del bus CAN. Este microcontrolador implementa la especificación CAN versión 2.0B. Está capacitado para transmitir y recibir tramas estándar, extendidas y remotas. El MCP2515 posee 2 máscaras de aceptación y seis filtros que son usados para filtrar mensajes no deseados, reduciendo así la carga de trabajo del microcontrolador. El MCP2515 se comunica vía SPI con el microcontrolador en el que se integra este shield.

Sus características básicas son:

- Implementa CAN V2.0B a 1 Mb/s:
    - 0-8 bytes en el campo de datos.
    - Tramas estándar, extendidas y remotas.
- Buffers de recepción, máscaras y filtros:
    - Dos buffers de recepción con prioridad en el almacenaje de tramas.
    - Seis filtros de 29 bits (en el caso de las tramas extendidas, en el caso de las tramas estándar será de 11 bits).
    - Dos máscaras de 29 bits (en el caso de las tramas extendidas, en el caso de las tramas estándar será de 11 bits).
- Filtros en los bytes de datos en los primeros dos bytes de datos (se aplica en las tramas estándar).
- Tres buffers de transmisión con capacidad de priorización y de abortar.
- SPI de alta velocidad (10MHz).
- Modo One-shot que garantiza que la transmisión del mensaje se realiza sólo una vez.
- Pin de salida de reloj con un prescaler programable:
    - Se puede usar como reloj fuente de otros dispositivos.
- Señal Start-of-Frame(SOF) está capacitada para monitorizar la señal SOF:
    - Se puede usar en los protocolos time-slot-based y/o diagnósticos del bus para detectar degradaciones en el bus tempranas.
- Pin de salida de interrupción con selección habilitada.
- Pin de salida de buffer lleno configurable para:
    - Interrupción de cada buffer de recepción.
    - Salida de propósito general.
- Pin de entrada individual Request-to-Send(RTS) configurable para:
    - Controlar el pin de retransmisión de cada buffer de transmisión.
    - Entradas de propósito general.
- Tecnología CMOS de bajo consumo:
    - Rango de operatividad desde 2.7V-5.5V.

    - 5mA de corriente activa.
    - 1 µA de espera activa (Sleep mode).
- Rangos de temperatura soportados:
    - Industria(I): -40ºC a 85ºC.
    - Extendido(E): -40ºC a 125ºC.

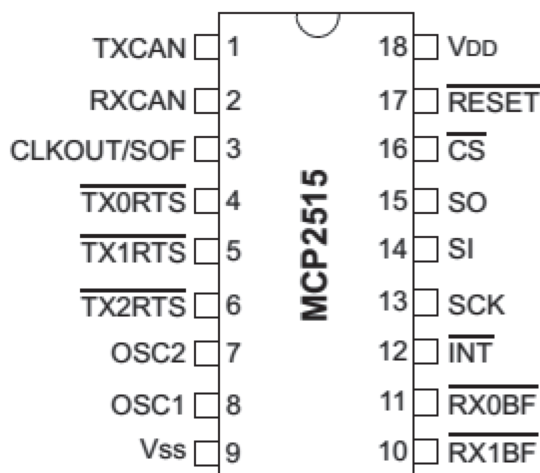


Ilustración 18: Esquema MCP2515. Fuente: [5]

Como hemos visto anteriormente, el MCP2515 implementa un controlador CAN integrado, y en este tipo de implementaciones el transceptor (transceiver) no está dentro del propio microcontrolador. En definitiva, para poder trabajar con él deberemos conectar uno de estos transceptores al microcontrolador.

El transceptor que usa este shield es el MCP2551[6] que puede direccionar una carga mínima de 45 Ohms, permitiendo conectarse a un máximo de 112 nodos. La entrada del transceptor, RXD, proporciona el diferencial de tensión entre CANH y CANL.

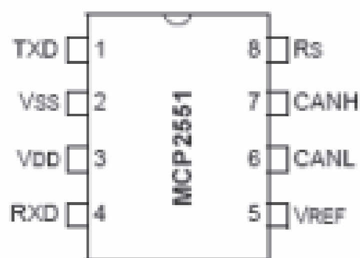


Ilustración 19: Esquema MCP2551. Fuente: [5]

El pin Rs permite elegir entre 3 modos de funcionamiento:

- **High Speed mode:** Se consigue conectando el pin Rs a Vdd.
- **Slope Control:** Reduce enormemente las emisiones electromagnéticas disminuyendo los tiempos de subida y de caída de CANH y CANL. Este control se logra mediante una resistencia entre Rs y Ground (masa). La tasa de ralentización es proporcional a la salida actual en Rs.
- **Standby mode:** Se consigue poniendo a nivel alto la patilla Rs. En este modo, el transmisor está apagado, y el receptor funciona ralentizado, es decir, el pin del receptor RXD seguirá operativo, pero de forma más lenta.

La conexión física sería a través de los pines RXCAN (recepción CAN del MCP2515), TXCAN (transmisión CAN del MCP2515), con los pines RXD y TXD del transceptor (MCP2551). Recordar que la función del transceptor es la de adecuar la señal de entrada del bus a la del sistema. Esta entrada de señal del bus se efectúa a través de los patillas CANH y CANL del transceptor.

Aquí vemos la conexión entre el MCP2515 y el MCP2551 representado por un diagrama de bloques:

Ilustración 20: Conexión entre MCP2515 y MCP2551 Fuente: [5] y [6]

4.1.7.2 Bloque de protocolo SPI

El microcontrolador se conecta al shield vía SPI. Escribiendo y leyendo de todos los registros usando los comandos estándar SPI de lectura y escritura, además de comandos SPI especializados.

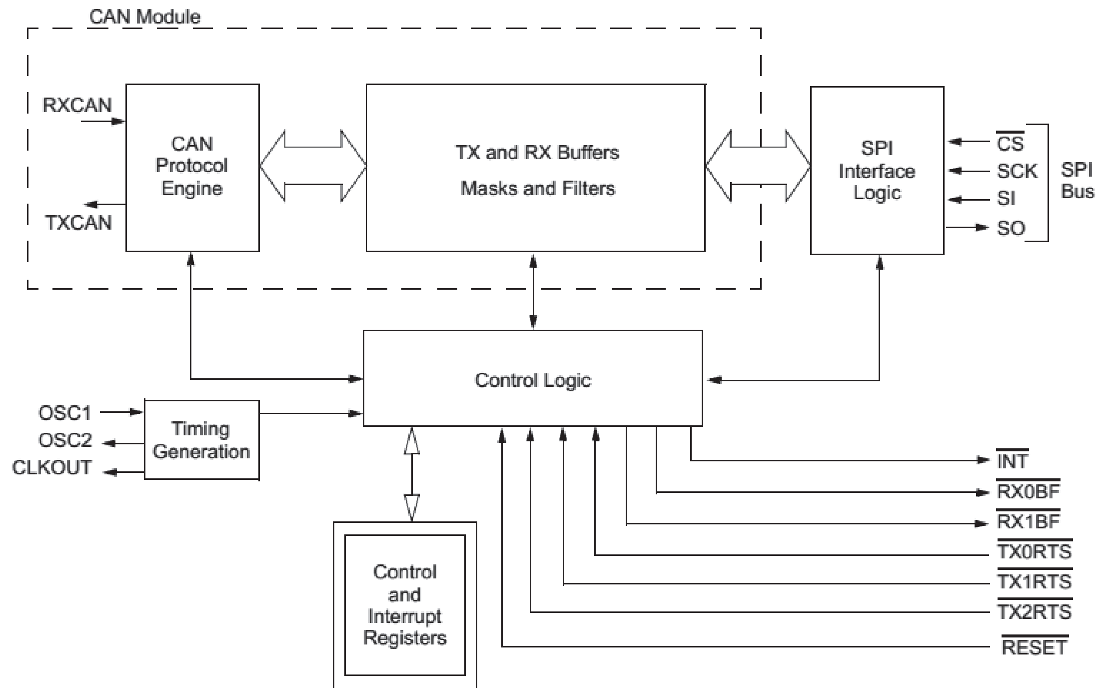


Ilustración 21: Bloque lógico general con el bloque SPI. Fuente: [5]

4.1.7.3 Modos de funcionamiento

El MCP2515 puede trabajar entre 5 modos de funcionamiento, previamente escogido por el usuario:

- **Configuration mode**
- **Normal mode.**
- **Sleep mode.**
- **Listen-only mode.**
- **Loopback mode**

El modo de operación se selecciona con los bits REQOP del registro CANCTRL.

Cuando se cambia de modo, no entrará en funcionamiento hasta que se complete la transmisión de todos los mensajes pendientes. El

modo pedido se debe verificar leyendo los bits OPMODE del registro CANSTAT.

Configuration mode

El MCP2515 tiene que ser inicializado antes de activarse. Esto sólo es posible si el dispositivo se encuentra en Configuration Mode. En este modo es seleccionado automáticamente antes de encenderse, resetearse o entrar en algún otro modo estableciendo los bits REQOP del registro CANCTRL. Cuando entra en este modo los contadores de error se reestablecen, y es el único modo en el que se modifican los registros CNF1, CNF2, CNF3, TXRTSCTRL, Registros de los filtros, y registros de las máscaras.

Normal mode

Es el modo de operación estándar del MCP2515. En este modo, el dispositivo monitoriza activamente todos los mensajes del bus y genera bits de reconocimiento, tramas de error, etc. Es también el único modo en el que el MCP2515 transmite mensajes por el bus.

Sleep mode

Este modo se usa para minimizar el consumo energético del dispositivo. La interfaz SPI permanece activa incluso cuando el MCP2515 se encuentra en este modo, permitiendo acceso a todos los registros. Para entrar en este modo hay que establecer los bits REQOP del registro CANCTRL con el correspondiente valor.

Cuando el microcontrolador se encuentra en este modo, la interrupción de wake-up esta activa (si se ha habilitado previamente). Con esta interrupción el MC2515 puede despertarse cuando detecte actividad en el bus.

En este modo el MCP2515 para su oscilador interno. Se despertará cuando ocurra actividad en el bus o cuando el microcontrolador establezca, vía SPI, el bit WAKIF del registro CANINTF para generar el wake-up (el bit WAKIE del registro CANINTE tiene que estar establecido para que se produzca la interrupción de wake-up).

El pin TXCAN se pone a recesivo cuando el MCP2515 se encuentra en este modo.

Listen-only mode

El modo Listen-only proporciona al MCP2515 a recibir todos los mensajes (incluidos los mensajes de error) configurando los bits RXM del registro RXBnCTRL. Este modo se puede usar para monitorizar aplicaciones en el bus o para detectar la velocidad del bus cuando se producen conexiones en caliente.

Es un modo silencioso, en la que no se transmiten mensajes mientras nos encontremos en este modo (incluyendo flags de error o señales de reconocimiento). Los filtros y máscaras se pueden poner a cero para permitir a las tramas con cualquier identificador ser leídas. Los contadores de error son reseteados y desactivados.

Loopback mode

Permite transmisiones internas de mensajes de los buffers de transmisión a los buffers de recepción sin actualizar la transmisión de mensajes en el bus. Este modo se usa para desarrollo y testeo de sistema. En este modo el bit ACK de las tramas es ignorado y el dispositivo habilita la recepción de mensajes de el mismo como si la hubiera recibido de otro nodo. Es un modo silencioso. El pin TXCAN estará en estado recesivo.

Los filtros y las máscaras se pueden usar para cargar mensajes particulares en los registros de recepción. Las máscaras pueden establecerse a 0 para aceptar todos los mensajes.

R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-1
REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0
bit 7							bit 0

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

bit 7-5 **REQOP:** Request Operation Mode bits <2:0>

- 000 = Set Normal Operation mode
- 001 = Set Sleep mode
- 010 = Set Loopback mode
- 011 = Set Listen-only mode
- 100 = Set Configuration mode

Ilustración 22: Registro CANCTRL. Fuente: [5]

R-1	R-0	R-0	U-0	R-0	R-0	R-0	U-0
OPMOD2	OPMOD1	OPMOD0	—	ICOD2	ICOD1	ICOD0	—
bit 7							bit 0

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

bit 7-5 **OPMOD**: Operation Mode bits <2:0>

- 000 = Device is in the Normal operation mode
- 001 = Device is in Sleep mode
- 010 = Device is in Loopback mode
- 011 = Device is in Listen-only mode
- 100 = Device is in Configuration mode

Ilustración 23: Registro CANSTAT. Fuente: [5]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF
bit 7							bit 0

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

Ilustración 24: Registro CANINTF. Fuente: [5]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE
bit 7							bit 0

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

Ilustración 25: Registro CANINTE. Fuente: [5]

Register Name	Address (Hex)	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	POR/RST Value
BFPCTRL	0C	—	—	B1BFS	B0BFS	B1BFE	B0BFE	B1BFM	B0BFM	--00 0000
TXRTSCTRL	0D	—	—	B2RTS	B1RTS	B0RTS	B2RTSM	B1RTSM	B0RTSM	--xx x000
CANSTAT	xE	OPMOD2	OPMOD1	OPMOD0	—	ICOD2	ICOD1	ICOD0	—	100- 000-
CANCTRL	xF	REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0	1110 0111
TEC	1C	Transmit Error Counter (TEC)								0000 0000
REC	1D	Receive Error Counter (REC)								0000 0000
CNF3	28	SOF	WAKFIL	—	—	—	PHSEG22	PHSEG21	PHSEG20	00-- -000
CNF2	29	BTLMODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0	0000 0000
CNF1	2A	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	0000 0000
CANINTE	2B	MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE	0000 0000
CANINTF	2C	MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF	0000 0000
EFLG	2D	RX1OVR	RX0OVR	TXBO	TXEP	RXEP	TXWAR	RXWAR	EWARN	0000 0000
TXB0CTRL	30	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-00
TXB1CTRL	40	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-00
TXB2CTRL	50	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	-000 0-00
RXB0CTRL	60	—	RXM1	RXM0	—	RXRTR	BUKT	BUKT	FILHIT0	-00- 0000
RXB1CTRL	70	—	RSM1	RXM0	—	RXRTR	FILHIT2	FILHIT1	FILHIT0	-00- 0000

Ilustración 26: Tabla resumen de los registros de control. Fuente: [5]

4.1.7.4 Recepción de mensajes

4.1.7.4.1 Buffers de recepción de mensajes

El MCP2515 incluye dos buffers de recepción completos con múltiples filtros de aceptación para cada uno. También existe un buffer de ensamblaje de mensajes (Message Assembly Buffer→MAB) que actúa como un tercer buffer de recepción.

De estos tres buffers de recepción, el MAB es siempre entregado a recibir el siguiente mensaje del bus, ensamblando todo el mensaje recibido. Estos mensajes serán trransferidos a los buffers RXBn solo si se cumple el criterio establecido en los filtros de aceptación.

Los otros dos buffers de recepción restantes, llamados RXB0 y RXB1, pueden recibir mensajes completos desde el MAB. El microcontrolador da acceso a un buffer, mientras el otro buffer está disponible para la recepción de mensajes, o para retener un mensaje anteriormente recibido.

Cuando un mensaje es aceptado todo el contenido que se encuentra en el MAB se mueve al buffer de recepción. Esto significa independientemente del tipo de identificador (estándar o extendido) y el número de bytes de datos recibidos, el buffer de recepción entero es sobrescrito en el contenido del MAB. Por lo tanto, el contenido de los registros del buffer es modificado cuando se recibe cualquier mensaje.

Cuando un mensaje es movido a cualquier buffer de recepción, el bit apropiado RXnIF del registro CANINTF. Es establecido. Este bit será borrado por el propio microcontrolador para poder recibir un nuevo mensaje en el buffer. Este bit es el que provee de un bloqueo positivo al microcontrolador para finalizar con cualquier mensaje antes de que el MCP2515 intente cargar un nuevo mensaje en el buffer de recepción. Si el bit RXnIE del registro CANINTE está establecido, se generará una interrupción en el pin INT' para indicar que se ha recibió un mensaje válido, así como el pin RXnBF asociado se pondrá a nivel bajo si está configurado como pin de buffer de recepción lleno.

4.1.7.4.2 Prioridad en la recepción de mensajes

El buffer RXB0 posee mayor prioridad y tiene una máscara y dos filtros de aceptación asociados, con lo que cuando se produce una recepción de un mensaje se aplica primero la máscara y los filtros del buffer RXB0. EL buffer RXB1 tiene menor prioridad y posee una máscara y cuatro filtros de aceptación.

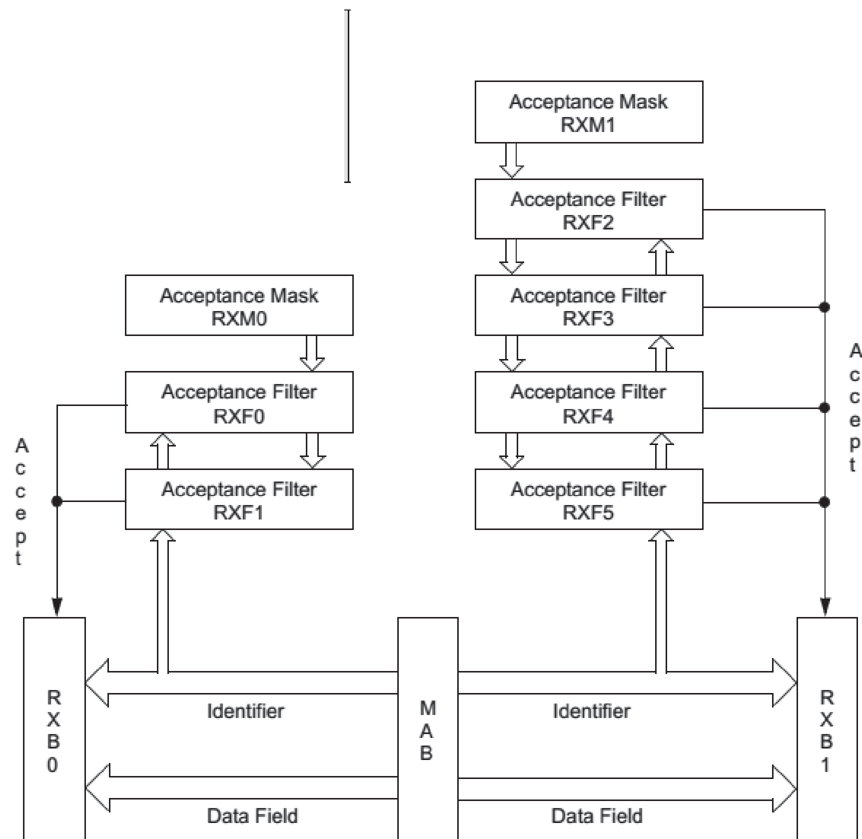


Ilustración 27: Diagrama de bloques de los buffers de recepción. Fuente: [5]

Cuando se recibe un mensaje los bits del 3 al 0 del registro RXBnCTRL indicarán el número del filtro de aceptación que habilito la recepción y si el mensaje recibido es una solicitud de transferencia remota.

El registro RXB0CTRL puede ser configurado para que, si el buffer RXB0 contiene un mensaje valido y se recibe otro mensaje válido, no se producirá un error de overflow y este nuevo mensaje se moverá el buffer RXB1, independientemente de los criterios de aceptación de RXB1. A este procedimiento se le conoce como ROLLOVER y se habilita con el bit BUKT del registro de control RXB0CTRL.

4.1.7.4.3 Filtros de aceptación de mensajes

También están los bits RXM de los registros RXBnCTRL para modos de recepción especiales. Normalmente estos bits son borrados a los valores 00 para habilitar la recepción de todos los mensajes validos que cumplan los criterios de los filtros de aceptación. En este caso, la determinación de recibir o no recibir mensajes estándar o extendidos la determinara el bit EXIDE de los registros de los filtros de aceptación RFXnSIDL. Si los valores de estos bits RXM están establecidos a 01 o 10, el receptor solo aceptara mensajes con identificadores estándar o extendidos respectivamente. Si el filtro de aceptación tiene establecido el bit EXIDE y no corresponde con el modo establecido en los bits RXM, el filtro de aceptación será inutilizado. Si el valor de los bits RXM está establecido a 11, el buffer aceptará cualquier mensaje, independientemente del valor de los filtros de aceptación. Si un mensaje posee un error antes del EOF, esa porción del mensaje será construida en el MAB antes de que el mensaje de error sea cargado en el buffer. Este modo tiene algunos valores de depuración en un sistema CAN y no debe ser usada en un entorno actual de sistemas.

Nota: La distribución de los de los registros de control nombrados en este apartado para la recepción de mensajes se encuentra en la ilustración 26. En el Anexo 1 se encuentra un diagrama de flujo en el que se contempla con más detalle como realiza el MCP2515 la recepción de mensajes.

4.1.7.5 Transmisión de mensajes

4.1.7.5.1 Buffers de transmisión de mensajes

El MCP2515 implementa tres buffers de transmisión. Cada uno de estos buffers ocupa 14 bytes de SRAM y están mapeados en el dispositivo de mapeo de memoria.

El primer byte es el registro de control TXBncTRL asociado al buffer de mensajes. La información en este registro determina las condiciones bajo las que el mensaje será transmitido indicando el estado de la transmisión del mensaje. 5 bytes son usados para soportar los identificadores estándar y extendidos, así como otra información de arbitraje del mensaje. Los últimos ocho bytes son para los 8 posibles bytes de datos que es capaz de transmitir el mensaje.

Como mínimo, tienen que estar cargados los registros TXBnSIDH, TXBnSIDL y TXBnDLC.Si los bytes de datos están presentes en el mensaje, los registros TXBnDm tienen que estar también cargados. Si el mensaje usa identificadores extendidos, los registros TXBnEIDm tienen que estar cargados, así como el bit EXIDE del registro TXBnSIDL.

Antes de enviar el mensaje, el microcontrolador debe inicializar los bits TXInE del registro CANINTE, para habilitar o deshabilitar las interrupciones cuando el mensaje sea enviado.

4.1.7.5.2 Prioridad en la transmisión de mensajes

Antes de enviar el Start-Of-Frame(SOF), se compara la prioridad de todos los buffers que están encolados para la transmisión. Primero envía el buffer que tenga la mayor prioridad. En el caso de que los dos buffers tengan la misma prioridad, el buffer con mayor número enviará primero.

Hay 4 niveles de prioridad al transmitir. Si los bits TXP de uno de los registros TXBnCTRL están establecidos a 11 el buffer tiene la mayor prioridad, si estos bits están establecidos a 00 el buffer tiene la menor prioridad.

4.1.7.5.3 Iniciar la transmisión de mensajes

Cuando se va a iniciar la transmisión de un mensaje el bit TXREQ de alguno de los registros TXBnCTRL tiene que estar establecidos para que cada buffer sea transmitido. Esto se puede realizar de distintas maneras.

- Escribiendo el registro por el comando SPI de escritura
- Enviando el comando SPI RTS
- Estableciendo el pin TXnRTS' a nivel bajo para el buffer particular que va a ser transmitido

Si la transmisión está iniciada por la interfaz SPI, el bit TXREQ se establece a la misma vez que los bits de prioridad TXP. Cuando el bit TXREQ de alguno de los registros TXBnCTRL, los bits ABTF, MLOA y TXERR del mismo registro son reseteados automáticamente. El establecer el bit TXREQ no inicia la transmisión, esta se iniciará cuando el dispositivo detecte que el bus está disponible.

Cuando la transmisión se completa satisfactoriamente, este bit TXREQ se resetea, y el bit TXnIF del registro CANINTF se habilita para que genere la interrupción si el bit TXnIE del registro CANINTE está habilitado.

Si la transmisión del mensaje falla el bit TXREQ permanecerá habilitado. Esto indica que el mensaje está pendiente de transmisión y alguno de los siguientes flags de condición estará habilitado:

- Si el mensaje inició su transmisión, pero se encontró un error de condición, el bit TXERR del registro del respectivo buffer TXBnCTRL y el bit MERRF del registro CANINTF se habilitan para generar una interrupción en el pin INT' si el bit MERRE del registro CANINTE está habilitado.
- Si el mensaje se ha perdido, el arbitraje deberá establecerse en el bit MLOA del registro TXBnCTRL del buffer respectivo.

Si el dispositivo se encuentra en el modo One-Sot las condiciones anteriores seguirán existiendo. Sin embargo, el bit TXREQ será borrado y el mensaje no reintentará por segunda vez reenviarse.

4.1.7.5.4 Abortar la transmisión de mensajes

El microcontrolador puede abortar la transmisión de un mensaje especifico restableciendo el bit TXREQ del registro TXBnCTRL asociado. También se puede establecer el bit ABAT del registro CANCTRL para abortar las transmisiones de todos los mensajes pendientes

Nota: La distribución de los de los registros de control nombrados en este apartado para la transmisión de mensajes se encuentra en la ilustración 26. En el Anexo 2 se encuentra un diagrama de flujo en el que se contempla con más detalle como realiza el MCP2515 la transmisión de mensajes.

4.1.7.6 Temporización y sincronización de bits

4.1.7.6 .1 Temporización

Todos los nodos conectados al bus CAN tienen que tener la misma tasa de bit nominal. El protocolo CAN usa la codificación NRZ (No Return to Zero) para codificar el reloj en el flujo de datos. El reloj de recepción debe recuperarse de la recepción de los nodos y sincronizarse para transmitir su señal de reloj.

Como los osciladores y los tiempos de transmisión pueden variar de un nodo a otro, el receptor debe tener algún tipo de PLL (Phase Lock Loop) sincronizando los niveles de transmisión de datos para sincronizar y mantener el reloj de recepción. Ya que los datos usan la codificación NRZ, es necesario que se incluya bit stuffing para asegurar que ocurra un cambio de nivel cada seis bits iguales para mantener la sincronización DPLL (Digital Phase Lock Loop).

La temporización del MCP2515 está implementada usando un DPLL configurado para sincronizar los datos recibidos, así como proveer tiempo nominal a los datos transmitidos. Este DPLL rompe cada temporización de bit en múltiples segmentos, obteniendo periodos de tiempo mínimos llamados quantums de tiempo (TQ).

Las funciones ejecutadas de temporización del bus con la temporización de los bits son definidas por la temporización lógica programable del DPLL.

Todos los dispositivos conectados al bus CAN deben usar la misma tasa de bits. Sin embargo, los dispositivos no están requeridos a tener la misma frecuencia del reloj del oscilador del maestro. Para las frecuencias de reloj diferentes de cada dispositivo, la tasa de bits tiene que ajustarse estableciendo un prescaler y un número de quantum de tiempo en cada segmento.

El tiempo de bit CAN se compone de segmentos sin solapamiento. Cada uno de estos segmentos se compone de quantums de tiempo(TQ). La tasa de bits nominal (Nominal Bit Rate(NBR)) está definida en la especificación CAN como el número de bits por segundo transmitidos sin re sincronización.

$$NBR = f_{bit} = \frac{1}{t_{bit}}$$

Ilustración 28: Ecuación tasa de bit nominal. Fuente: [5]

El tiempo de bit nominal (Nominal Bit Time (NBT o t_{bit})) también se compone de segmentos sin solapamiento.

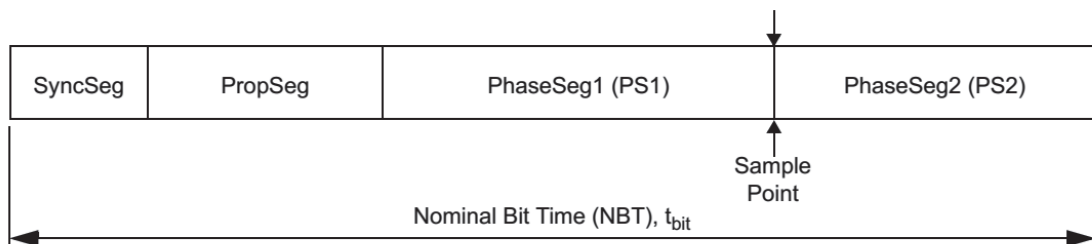


Ilustración 29: Segmentos de tiempo de bit CAN. Fuente: [5]

Por lo tanto, este tiempo es el sumatorio de los segmentos siguientes:

$$t_{bit} = t_{SyncSeg} + t_{PropSeg} + t_{PS1} + t_{PS2}$$

Ilustración 30: Ecuación tiempo de bit nominal. Fuente: [5]

Asociado al tiempo de bit nominal está el punto de muestreo, salto de sincronización de anchura (Synchronization Jump Width (SJW)) y el tiempo de proceso de información (Information Processing Time (IPT)).

Segmento de sincronización (SyncSeg): El segmento de sincronización es el primer segmento en el tiempo de bit nominal y se usa para sincronizar nodos en el bus. Se esperan cambios de nivel lógico dentro de este segmento. Está fijado a 1 quantum de tiempo.

Segmento de propagación (PropSeg): El segmento de propagación existe para compensar los retrasos físicos entre nodos. La propagación de este retraso está definida como dos veces la suma de la señal de tiempo de propagación en el bus, incluyendo los retrasos asociados al driver del bus. El tiempo de propagación es programable de 1 a 8 quantums de tiempo.

Segmento de fase 1 (PS1) y segmento de fase 2 (PS2): Estos dos segmentos de fase se usan para compensar los errores de cambios de fase en el bus. PS1 puede alargarse (o PS2 acortarse) por re sincronización. PS1 es programable desde 1 a 8 quantums de tiempo mientras que PS2 es programable de 2 a 8 quantums de tiempo.

Punto de muestreo: El punto de muestreo es el punto en el tiempo de bit en el que se lee e interpreta el nivel lógico. Se encuentra al final de PS1. La ejecución de esta regla es si el modo de muestra está configurado para muestrear tres veces por bit. En este caso, mientras el bit esté aún muestreado al final de PS1, se producen dos muestreos adicionales a mitad de los intervalos de los quantums de tiempo antes del final de PS1, con el valor del bit determinado por decisión mayoritaria.

Tiempo de proceso de información (IPT): El tiempo de proceso de información es el tiempo requerido por la lógica para determinar el nivel de un bit que se ha muestreado. Este IPT comienza con un punto de muestreo, es medido en quantums de tiempo y está fijado a 2 por el módulo CAN del microchip. Ya que PS2 también empieza en el punto de muestreo y es el último segmento en el tiempo de bit, es requerido que el PS2 mínimo no sea menor que el IPT.

Salto de sincronización de anchura (SJW): El salto de sincronización de anchura ajusta el reloj de bit como sea necesario de 1 a 4 quantums de tiempo para mantener la sincronización con el mensaje transmitido.

Quantum de tiempo: Cada uno de estos segmentos que constituye un tiempo de bit determinado está compuesto de unidades de enteros

llamadas quantums de tiempo (TQ). La longitud de cada quantum de tiempo está basada en el periodo del oscilador (t_{OSC}). El quantum de tiempo es igual a dos veces el periodo del oscilador y la longitud de este es una vez el periodo de reloj de dicho quantum(t_{BRPCLK}), que es programable usando un prescaler programable llamado Prescaler de tasa de transmisión (Baud Rate Prescaler (BDP)).

$$TQ = 2 \cdot BRP \cdot T_{OSC} = \frac{2 \cdot BRP}{F_{OSC}}$$

Ilustración 31: Ecuación para calcular el Quantum de Tiempo(TQ). Fuente: [5]

4.1.7.6.2 Sincronización

Para compensar los desplazamientos de fase entre la frecuencia del oscilador de cada uno de los nodos, cada controlador CAN debe estar sincronizado al flanco de la señal de entrada. La sincronización es el proceso por el que el DPLL está implementado.

Cuando se detecta un flanco en los datos transmitidos, la lógica comparará la localización de este flanco con el tiempo esperado (SyncSeg). El circuito ajustará luego PS1 y PS2 como sea necesario.

El MCP2515 posee dos mecanismos de sincronización:

- Sincronización de trama (Hard Synchronization).
- Resincronización.

Sincronización de trama (Hard Synchronization): Esta sincronización solo se produce cuando hay un cambio de nivel lógico de recesivo a dominante mientras se cumple la condición de BUS IDLE, indicada por el comienzo e un mensaje. Tras esta sincronización, el contador de tiempo de bit es reiniciado con SyncSeg.

Resincronización: Como resultado de esta resincronización, PS1 puede ser alargado o PS2 acortado. La cantidad en la que se alarga o se acorta los segmentos de fase tienen un límite superior dados por el salto

de sincronización de anchura (SJW). El valor de este SJW se añadirá a PS1 sustraído de PS2.

4.1.7.6.3 Programación de los segmentos de tiempo y configuración de los registros

Algunos requerimientos para programar los segmentos de tiempo son:

- PorpSeg + PS1 >= PS2
- PropSeg + PS1 >= T_{DELAY}
- PS2 > SJW

La configuración para la temporización de los bits se realiza a través de los registros CNF1, CNF2 y CNF3. Estos registros Solo pueden ser modificados cuando el MCP2515 se encuentre en modo configuración.

CNF1

Los bits BRP controlan el prescaler de la tasa de transmisión. Estos bits establecen la longitud de los TQ´s relativos a la entrada de frecuencia del OSC1, con la mínima longitud de TQ que será 2 veces T_{OSC}. Los bits SJW seleccionan el SJW en términos de número de TQ´s.

CNF2

Los bits PRSEG establecen la longitud (en quantums de tiempo (TQ)) de la propagación del segmento. Los bits PHSEG establecen la longitud en TQ del segmento PS1.

El bit SAM controla cuantas veces el pin RXCAN es muestreado. Estableciendo este bit a 1 el bus es muestreado tres veces (dos veces en TQ/2 y otra en el punto de muestreo normal (al final de PS1)). El valor obtenido del bus se determina por la mayoría de estos muestreos. Si este bit está a 0 el pin RXCAN solo se muestreará una vez en el punto de muestreo normal.

El bit BTLMODE controla cuanta longitud se establece para PS2. Si el bit se encuentra a 1, la longitud de PS2 se determina por los bits PHSEG del registro CNF3. Si está puesto a 0, la longitud de PS2 es mayor

que la de PS1 y el tiempo de procesamiento de la información (que estará fijado a 2 por defecto por el MCP2515).

CNF3

Los bits PHSEG establecen la longitud de PS2, si el bit BTLMODE del registro CNF2 está establecido a 1, sino, no tiene efecto alguno.

Nota: La distribución de los de los registros de control nombrados en este apartado para configuración de los segmentos de tiempo se encuentra en la ilustración 26.

4.1.8 Librería ArCan

La librería ArCan se presenta como un proyecto de fin de carrera presentado por el alumno Raúl Milla Pérez para la Universidad de Málaga [16]. Esta librería está desarrollada para el manejo y configuración para Arduino del MCP2515 en el envío y recepción de tramas CAN a través de este tipo de bus.

A esta librería se le han añadido mejoras para poder desarrollar este TFG, ya que, la versión escogida de esta librería fue la 1.0 y presentaba limitaciones tanto de configuración como de funcionalidad.

Toda la funcionalidad de ArCan es accesible mediante la clase tArCan. En la parte pública se definen todos los métodos a funciones básicas para el uso de ArCan, en ellos se implementa lo necesario para enviar/recibir cualquier mensaje con el estándar CAN 2.0A y CAN 2.0B. La parte privada de la clase no es accesible desde las aplicaciones que podamos hacer en Arduino, pero con el conocimiento suficiente del hardware podemos realizar configuraciones más concretas.

Lo primero de la declaración de la clase tArCan es la definición de registro tCAN, este registro contiene los parámetros de un mensaje CAN 2.0 B:

```
typedef struct{
    struct{
        uint32_t id;
        byte ide : 1;
        byte rtr : 1;
    }arb_field;

    byte length : 4;

    byte data [8];
}tCAN;
```

Ilustración 32: Registro librería ArCan

Como podemos ver en la estructura se detallan todas las partes configurables de un mensaje CAN 2.0B. El uso de este tipo de tramas en el desarrollo de este TFG es que, al tener más bits en el campo de arbitraje, cabe la posibilidad de introducir más variables para tener una mayor capacidad de control en los datos, esto lo veremos en las secciones posteriores, concretamente cuando se hable de la librería SlidWind.

Dentro de la variable id, como es de 32 bits, aprovechamos para introducir los siguientes parámetros:

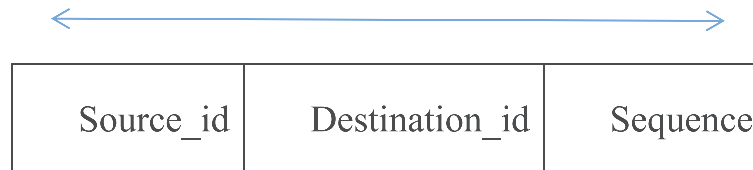
29 bits

Source_ id	Destination _id	Type_ of	Sequen ce	Sequence_a ck

1. Id del emisor (source_id)→10 bits.
2. Id del receptor (destination_id)→ 10 bits.
3. Tipo del mensaje (type_of)→ 3 bits.
4. Número de secuencia del mensaje (sequence)→ 3 bits.
5. Número de confirmación de secuencia del mensaje (sequence_ack)→ 3 bits.

Como vemos completamos así los 29 bits del campo de arbitraje que usan las tramas extendidas.

Para las tramas estándar que, aunque no las vamos a usar siguen la siguiente distribución:



1. Id del emisor (source_id)→4 bits.
2. Id del receptor (destination_id)→ 4 bits.
3. Numero de secuencia (sequence)→ 3 bits.

Completando así los 11 bits del campo de arbitraje que usan las tramas estándar.

También tendremos la variable packet_type que nos indicará el tipo del paquete, tendremos 3 tipos diferentes. La funcionalidad de esta variable la veremos en apartados posteriores, concretamente, en el protocolo de ventana deslizante.

enum packet_type {data, ack, nack};

A continuación, se presenta todas las funciones de la parte pública, accesibles por el usuario, que estaban ya implementadas y otras que se implementaron en esta librería para el desarrollo de este TFG.

boolean init (uint32_t filter, uint32_t mask_ext, uint32_t mask_std):

Con la llamada a esta función inicializamos el dispositivo ArCan al que le pasamos el filtro que será id que le vamos a asignar al dispositivo, su máscara extendida y su máscara estándar. Para el desarrollo de este TFG sólo nos interesa la máscara extendida ya que vamos a enviar solo tramas extendidas, aunque el dispositivo estará capacitado para enviar tramas estándar.

En esta función se inicializa el puerto SPI ya que el acceso a los registros del MCP2515 se hace a través de este protocolo como se explica anteriormente. Se hace un reset por software a dicho microcontrolador y después del reset entramos en modo configuración.

En este modo de configuración, establecemos los valores de ls registros CNF1, CNF2 y CNF3. Así como los registros BFPCTRL, TXRTSCTRL (para configurar TXnTRS como entradas digitales), RXB0CTRL y RXB1CTRL (Activando los filtros en la recepción para mensajes válidos tanto estándar como extendidos y permitiendo el ROLLOVER).

Luego establecemos los registros pertinentes a los filtros y máscaras para permitir la entrada de tramas extendidas dirigidas a nuestro dispositivo. Estos registros son RXF0SIDH, RXF1SIDH, RXF2SIDH, RXF3SIDH, RXF4SIDH, RXF5SIDH y sus sucesivos, así como RXM0SIDH y RXM1SIDH y sus sucesivos.

Finalmente pasamos a modo normal y devolvemos verdadero si toda la configuración se ha producido correctamente.

boolean check_message(void):
Nos informa si existe algún mensaje CAN en los buffers de recepción devolviendo verdadero en caso de que exista algún mensaje o falso en caso contrario.

boolean check_free_buffer(void):
Este método nos informa si existe algún buffer libre de transmisión. Esto se realiza verificando todos los buffers de transmisión y comprobando que el buffer está vacío.

byte get_message(tCAN *msje):

Nos permite capturar los mensajes de los buffers de recepción para permitir su posterior procesamiento. Nos devuelve el registro de estado y obtenemos el mensaje leyendo vía SPI y almacenando estas lecturas en el mensaje (msje) en caso de que detecte alguno en los buffers.

Esto se realiza según el registro estado vamos obteniendo si hay algo en los buffers de recepción, en caso afirmativo vamos obteniendo los respectivos datos y devolvemos el registro estado, en caso negativo devolvemos 0.

byte send_message(tCAN* msje):

Nos permite enviar mensajes. Devuelve un byte que será la dirección del buffer por el que se envió el mensaje, en caso de que no haya podido enviarlo por estar todos los buffers ocupados nos devolverá 0.

Leemos si alguno de los bits 2, 4, y 6 del registro de estado se encuentran a 0. Estos bits se corresponden con el estado de los buffers de transmisión, de forma que, si alguno de estos bits se encuentra a 1 indicará que dicho buffer está ocupado, y si está a 0 que están libres. Luego vamos enviando vía SPI los parámetros del mensaje.

void print_message(tCAN *msje):

Nos muestra por el monitor serie el contenido del mensaje que le pasamos por parámetro, tanto mensajes estándar como extendidos. En nuestro caso solo mostraremos mensajes extendidos que son los que usamos en nuestra aplicación.

Mostramos su id, que es de 32 bits con sus respectivos campos vistos anteriormente, la longitud de los datos, el campo rtr que nos dice si es una trama remota o no, y el campo ide que nos indica si son tramas estándar o extendidas, y los datos.

void mode_loopback(void):

Este método pone nuestro dispositivo CAN en modo loopback. Como se expone en apartados anteriores, este método es puramente de testeo y depuración ya que vamos obteniendo en los buffers de recepción los mismos mensajes que vamos introduciendo en los buffers de transmisión. Esto lo realiza escribiendo vía SPI los bits correspondientes del registro CANCTRL para establecer este modo y verificando que los ha escrito leyendo vía SPI los bits correspondientes del registro CANSTAT.

void mode_normal():

Este método pone nuestro dispositivo CAN en modo normal. Como se expone en apartados anteriores, este método nos permite enviar y recibir mensajes a través del bus, y siempre que estemos en modo loopback o nos despertemos del modo dormido tendremos que pasar a

este modo de funcionamiento para hacer uso de forma normal del bus CAN. Esto lo realiza escribiendo vía SPI los bits correspondientes del registro CANCTRL para establecer este modo y verificando que los ha escrito leyendo vía SPI los bits correspondientes del registro CANSTAT.

void mode_sleep():

Este método pone nuestro dispositivo CAN en modo dormido. Como se expone en apartados anteriores, este método nos permite ahorrar energía despertando nuestro dispositivo cuando detecte actividad en el bus. Esto lo realiza escribiendo vía SPI los bits correspondientes del registro CANCTRL para establecer este modo y verificando que los ha escrito leyendo vía SPI los bits correspondientes del registro CANSTAT. Y habilitando la interrupción de wake-up para poder despertarnos de este modo, esto se realiza escribiendo vía SPI el bit WAKIE a 1 del registro CANINTE. Cuando nos despertamos de este modo, por defecto nos despertamos en el modo listen-only, y debemos pasarnos a modo normal para hacer un uso normal del bus, este cambio de modos se realiza en la función get_message ya que será aquí cuando estaremos escuchando el tráfico del bus.

void set_filter(uint32_t filter):

Aquí escribimos los registros pertinentes a los filtros extendidos vía SPI, teniendo en cuenta que el campo que hará de filtrado será de 11 bits tal y como estamos estableciendo los identificadores emisor y destino en el campo id de los mensajes. Mostramos por el monitor serial el valor de estos registros para que el usuario vea que se escribieron correctamente.

void set_mask(uint32_t mask_ext, uint32_t mask_std):

Escribimos vía SPI los registros pertinentes a las máscaras estableciéndolas a sus valores pasados por parámetro. En nuestro caso establecemos las dos máscaras al valor del parámetro mask_ext ya que, aunque se pueden usar tramas tanto estándar como extendidas nosotros solo vamos a usar tramas extendidas. Mostramos por el monitor serial el valor de estos registros para que el usuario vea que se escribieron correctamente.

uint32_t pack_ext(uint16_t source_id, uint16_t destination id, packet_type type of, uint8_t sequence, uint8_t sequence_ack):

En esta función metemos en una variable de 32 bits todas las variables pasadas por parámetro que se corresponderán con la distribución elegida para el campo de arbitraje (los 29 bits de las tramas extendidas).

void unpack_ext(uint32_t buffer,uint16_t& source_id, uint16_t& destination_id, packet_type& type_of, uint8_t& sequence, uint8_t& sequence_ack):

Obtenemos de la variable buffer, que se corresponderá con el campo de arbitraje de las tramas extendidas, el resto de campos almacenando cada uno de ellos en sus respectivas variables.

uint32_t pack_stand(uint16_t source_id, uint16_t destination_id, uint8_t sequence):

En esta función metemos en una variable de 32 bits todas las variables pasadas por parámetro que se corresponderán con la distribución elegida para el campo de arbitraje (los 11 bits de las tramas estándar).

void unpack_stand(uint32_t buffer, uint16_t& source_id, uint16_t& destination_id, uint8_t& sequence):

Obtenemos de la variable buffer, que se corresponderá con el campo de arbitraje de las tramas estándar, el resto de campos almacenando cada uno de ellos en sus respectivas variables.

packet_type get_type(uint8_t type_of):

Esta función nos devuelve según el valor del parámetro type_of, que indicará el tipo de paquete en la ventana deslizante explicada en secciones posteriores. Seguirá la correspondencia:

0. Datos
1. ACK
2. NACK

A continuación, se presentan todas las funciones de la parte privada, no accesibles por el usuario, pero de igual importancia para el funcionamiento de la librería ya que las funciones anteriormente de la parte pública hacen uso de estas. En general son las funciones que realizan la comunicación vía SPI:

byte spi_putc(byte data):

Escribe un byte pasado por el parámetro data por el bus SPI.

void write_register(byte direction,byte data):

Escribe un registro del MCP2515, estableciéndose primero en la dirección dada por el parámetro direction, y luego escribiendo el valor dado por el parámetro data en dicha dirección.

void write_registers(byte direction, byte values[], const byte length):

Escribe varios registros del MCP2515, estableciéndose primero en la dirección dada por el parámetro direction, y luego se recorre el vector de datos values obteniendo el dato y escribiéndolo en dicha dirección y sus consecutivas. Esto se hace tantas veces como valor contenga el parámetro length.

byte read_register(byte direction):

Lee un registro del MCP2515 obteniendo los datos que se encuentran en la dirección dada por el parámetro direction.

Void read_registers(byte direction, byte values[], const byte length):

Lee varios registros del MCP2515, estableciéndose primero en la dirección dada por el parámetro direction, y luego asignándole los datos obtenidos de dicha dirección al vector values. Esto se hace tantas veces como valor contenga el parámetro length.

Void bit_modify(byte direction, byte mask, byte data):

Modifica un bit de un registro del MCP2515. Nos establecemos en la dirección dada por el parámetro direction, y modificamos los bits

establecidos en el parámetro mask, con los valores establecidos en el parámetro data.

byte read_status(byte type):

Consulta el registro de estado del MCP2515.

4.2 Comunicación Serie

El módulo, unidad o puerto UART (Universal Asynchronous Receiver-Transmitter) es el dispositivo que controla los puertos y dispositivos serie asíncronos (RS232, etc.). El módulo UART toma bytes de datos y transmite los bits individuales de forma secuencial. En el destino, el módulo UART receptor reensambla los bits en bytes completos. La transmisión serie de la información digital (bits) a través de un cable único u otros medios es mucho más efectiva en cuanto a costo que la transmisión en paralelo a través de múltiples cables. Se utiliza un UART para convertir la información transmitida entre su forma secuencial y paralela en cada terminal de enlace. Cada UART contiene un registro de desplazamiento que es el método fundamental de conversión entre las forma secuencial y paralela. Usa dispositivos de interfaz separados para convertir las señales de nivel lógico del UART hacia y desde los niveles de señalización externos. Las señales externas pueden ser de variada índole. Ejemplos de estándares para señalización por voltaje son RS-232, RS-422 y RS-485 de la EIA.

En este TFG se usa el monitor serie que se utiliza para la comunicación entre la placa Arduino y una computadora u otros dispositivos. Todas las placas Arduino tienen al menos un puerto serie (también conocido como UART o USART). Se comunica en los pines digitales 0 (RX) y 1 (TX), así como con el ordenador a través de USB.

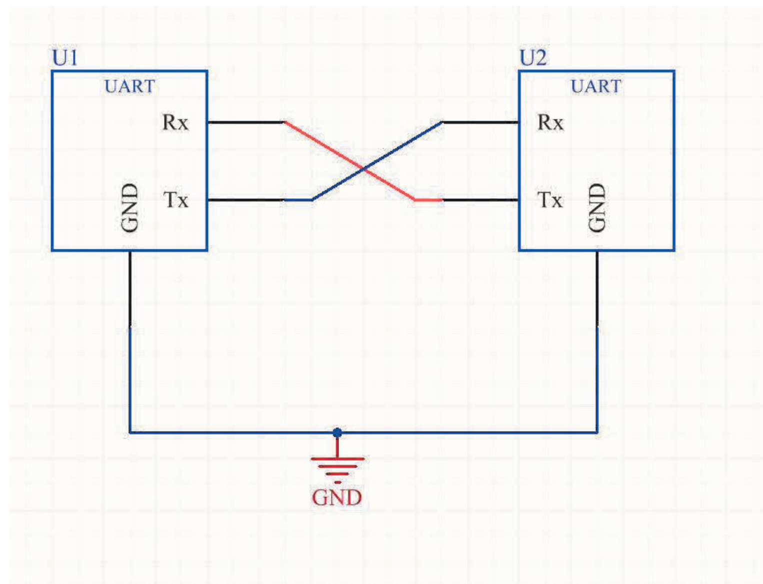


Ilustración 33: Comunicación entre dos dispositivos UART. Fuente: [14]

4.2.1 Librería Serial

La librería Serial [14] es la encargada de realizar la comunicación, manejo y funcionamiento del monitor serie accesible desde cualquier dispositivo Arduino. Es derivada de la clase Stream de Arduino con lo que puede usar la gran mayoría de las funciones de esta clase.

Toda la funcionalidad de la librería Serial es accesible mediante la clase Serial. A continuación, se presenta todas las funciones accesibles por el usuario, algunas siendo usadas para el desarrollo de este TFG.

if (Serial):

Indica si el puerto serie especificado está listo.

Serial.available():

Devuelve la cantidad de bytes (caracteres) disponibles para leer desde el puerto serie. Se trata de datos que ya han llegado y se almacenan en el búfer de recepción en serie (que contiene 64 bytes).

Serial.availableForWrite ():

Devuelve la cantidad de bytes (caracteres) disponibles para escribir en el buffer serie sin bloquear la operación de escritura.

Serial.begin (speed):

Establece la velocidad de datos en bits por segundo (baudios) para la transmisión de datos en serie. Para comunicarse con la computadora, se usa alguno de estos valores: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 115200 o 250000. No obstante, puede especificar otros valores, por ejemplo, comunicarse a través de los pines 0 y 1 con un componente que requiere una velocidad en baudios particular.

Serial.end():

Desactiva la comunicación serie, permitiendo que los pines RX y TX se usen para entrada y salida general. Para volver a habilitar la comunicación en serie, se llama a Serial.begin().

Serial.find(char target):

Lee datos del buffer serie hasta que se encuentra la cadena "target" de longitud dada. La función devuelve verdadero si se encuentra la cadena objetivo, falso si se agota el tiempo.

Serial.findUntil(char target, char terminal):

Lee los datos del buffer serie hasta que se encuentra una cadena de destino de longitud dada (target) o cadena de terminación (terminal).

La función devuelve verdadero si se encuentra la cadena objetivo o la cadena terminal y falso si se agota el tiempo.

Serial.flush():

Espera a que se complete la transmisión de los datos en serie salientes.

Serial.parseFloat():

Devuelve el primer número en coma flotante válido del buffer en serie. Los caracteres que no son dígitos (o el signo menos) se saltan. parseFloat(). Acaba por el primer carácter que no es un número de punto flotante.

Serial.parseInt() ó Serial.parseInt(char skipChar):

Devuelve el primer número entero del buffer en serie. En particular:

*Los caracteres iniciales que no son dígitos o un signo menos, se saltan.

*El análisis se detiene cuando no se han leído caracteres para un valor de tiempo de espera configurable, o no se lee un dígito.

*Si no se leyeron dígitos válidos cuando se acabe el tiempo de espera (vea Serial.setTimeout ()), se devuelve 0.

La variable pasada por parámetro skipChar se usa para que se salte en la búsqueda este carácter.

Serial.peek():

Devuelve el siguiente byte (carácter) de los datos en serie entrantes sin eliminarlo del buffer serie interno. Es decir, las llamadas sucesivas peek() devolverán el mismo carácter, al igual que la siguiente llamada a read() pero sin eliminarlo del buffer.

Serial.print(val) ó Serial.print(val, format):

Imprime datos en el puerto serie como texto ASCII. Este comando puede tomar muchas formas. Los números se imprimen utilizando un carácter ASCII para cada dígito. Los flotantes se imprimen de manera similar como dígitos ASCII, por defecto a dos decimales. Los bytes se envían como un solo carácter. Los caracteres y las cadenas se envían tal cual.

Un segundo parámetro opcional especifica la base (formato) a usar. Los valores permitidos son: BIN (binary, o base 2), OCT (octal, o base 8), DEC (decimal, o base 10), HEX (hexadecimal, o base 16). Para números en coma flotante, este parámetro especifica el número de decimales que se usarán.

Serial.println(val) ó Serial.println(val, format):

Imprime datos en el puerto serie como texto ASCII seguido de un carácter de retorno de carro (ASCII 13, o '\ r') y un carácter de nueva línea (ASCII 10 o '\ n'). Este comando toma el mismo esquema que Serial.print ().

Serial.read():

Lee los datos entrantes del buffer serie.

Serial.readBytes(char [] buffer, int length):

Lee caracteres del puerto serie almacenándolos en el parámetro buffer. La función termina cuando se leen caracteres hasta una longitud especificada por el parámetro length, o se cumple un determinado tiempo establecido previamente por Serial.setTimeout(). Retorna el número de caracteres almacenados en el parámetro buffer que será como máximo lo establecido en el parámetro length, o 0 en el caso de que no haya obtenido datos válidos.

Serial.readBytesUntil(char character, char [] buffer, int length):

Esta función va leyendo caracteres del buffer serie almacenándolos en el parámetro buffer. Esta función termina cuando se encuentra el parámetro character en el buffer serie, cuando se obtienen bytes hasta una determinada longitud especificada por el parámetro length, o cuando se cumple un determinado tiempo establecido previamente por Serial.setTimeout(). Retorna el número de caracteres almacenados en el parámetro buffer que será como máximo lo establecido en el parámetro length, o 0 en el caso de que no haya obtenido datos válidos.

Serial.setTimeout(long time):

Establece el número de milisegundos que espero por datos en el buffer serie. Si no se le pasa ningún parámetro posee por defecto 1000 milisegundos.

Serial.write(byte val) ó Serial.write(String str) ó Serial.write(char [] buf, int len):

Escribe datos en el buffer serie. Estos datos se envían al puerto serie como byte o array de bytes.

4.3 Protocolo de ventana deslizante

La ventana deslizante es un algoritmo o mecanismo software de control de flujo de datos en un canal bidireccional de comunicaciones. En él, el control del flujo se lleva a cabo mediante el intercambio específico de caracteres o tramas de control, con los que el receptor indica al emisor cuál es su estado de disponibilidad para recibir datos.

Este protocolo es necesario para no inundar al receptor con envíos de tramas de datos. El receptor al recibir datos debe procesarlos, si no lo realiza a la misma velocidad que el transmisor los envía, se verá saturado de datos y parte de ellos se podrían perder. Para evitar tal situación el algoritmo de la ventana deslizante controla este ritmo de envíos del emisor al receptor. Por otro lado, el algoritmo trata de sacar el máximo provecho del ancho de banda del canal de comunicaciones utilizado.

Con la implementación de este protocolo se resuelven dos grandes problemas: el control de flujo de datos y la eficiencia en la transmisión.

4.3.1 Funcionamiento ventana de transmisión

El protocolo de ventana deslizante permite al emisor transmitir múltiples segmentos de información antes de comenzar la espera para que el receptor le confirme la recepción de los segmentos, tal confirmación se llama validación, y consiste en el envío de mensajes denominados ACK del receptor al emisor. La validación se realiza desde el receptor al emisor y contiene el número de la siguiente trama que espera recibir el receptor, o el de la última trama recibida con éxito, ACK n (siendo n el número de la trama indicada). Con esta indicación el emisor es capaz de distinguir el número de los envíos realizados con éxito, los envíos perdidos y envíos que se esperan recibir.

Los segmentos Unacknowledge (UNACK) son segmentos que si han sido enviados, pero no han sido validados.El número de segmentos que pueden ser UNACK en un momento dado está limitado por el tamaño

de la ventana, un número pequeño y fijo, que se denomina el ancho de ventana.

El transmisor deberá guardar en un buffer todas aquellas tramas enviadas y no validadas (tramas UNACK), por si necesitase retransmitirlas. Sólo puede borrarlas del buffer al recibir su validación procedente del receptor, y deslizar así la ventana una unidad más. El número más pequeño de la ventana deslizante corresponde al primer paquete de la secuencia que no ha sido validado. El tamaño del buffer debe ser igual o mayor al tamaño de la ventana. El número máximo de tramas enviadas sin validar es igual al ancho de la ventana. De esta forma el buffer podrá almacenar temporalmente todas las tramas enviadas sin validar.

A cada uno de los segmentos pertenecientes al buffer (aquellos enviados y no validados), se les asigna un temporizador. El temporizador es el límite de tiempo de espera para recibir la validación de un determinado paquete. Si el paquete se pierde en el envío, el emisor nunca recibiría validación. El paquete nunca llegaría al receptor, este continuaría a la espera de recibir el paquete perdido. De esta manera el temporizador expiraría, tomando la decisión de reenviar la trama asignada al temporizador consumido. A este proceso se le conoce como "Stop and Wait".

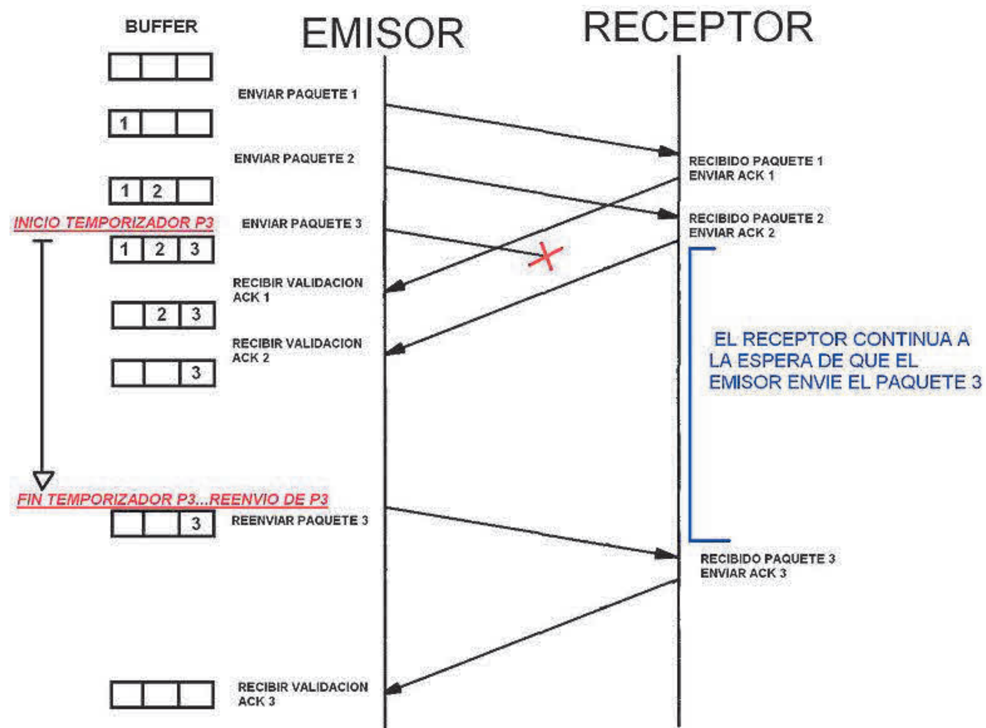


Ilustración 34: Funcionamiento ventana deslizante emisor con temporizadores. Fuente: Wikipedia

4.3.2 Funcionamiento ventana de recepción

El receptor posee una ventana de recepción, similar a la de transmisión, pero con una finalidad totalmente distinta. Su funcionalidad permite al receptor recibir un conjunto de tramas que le llegan desordenadas. La ventana de recepción es la lista que tiene el receptor con los números de la secuencia consecutivos de las tramas que puede aceptar. Almacena las tramas temporalmente en un buffer hasta el momento que posea todas las tramas esperadas, la secuencia de tramas esperada al completo, y así ordenarlas. El receptor debe disponer de un buffer de igual tamaño que su ventana de recepción para almacenar temporalmente las tramas hasta ordenarlas.

Existen 2 modos de trabajo en función del tamaño de su ventana:

- Tamaño ventana recepción=1→La ventana de recepción dispone de un buffer. Sólo puede almacenar la trama que le llega en cada instante, es decir, debe recibir las tramas en la secuencia

correcta, ya que no dispone de recursos para ordenarlas después. Impone al emisor la condición de transmitir siempre las tramas en secuencia.

- Tamaño ventana recepción>1. La ventana de recepción dispone de N buffers (N=tamaño ventana de recepción) que le permiten recibir hasta N tramas desordenadas, almacenarlas y proceder a su ordenamiento posterior. Le permite al emisor transmitir tramas desordenadas, tantas como quepan en los buffers del receptor.

4.3.3 Recuperación de errores

Existen 2 estrategias diferentes para la recuperación de errores:

Estrategia de rechazo simple (retroceso N, vuelta atrás, pullback NACK):

Tamaño ventana recepción=1. El receptor rechaza todas las tramas recibidas a partir de detectar una trama con error en el número de secuencia. Al detectar la trama errónea envía una señal REJ n (señal propia para este tipo de estrategia, n=nº trama errónea), o NACK n, (trama n no validada, n=nº trama errónea), al emisor para indicarle la situación. En ese instante el emisor comienza con la retransmisión de todas las tramas descartadas por el receptor, tanto la trama errónea como las tramas enviadas después de la trama errónea. Estas tramas retransmitidas por el transmisor se encontraban en el buffer del transmisor a espera de validación (tramas UNACK). No es un método efectivo, pierde mucho tiempo en la retransmisión.

Estrategia de rechazo selectivo (repetición selectiva, selective repeat):

Tamaño ventana recepción>1. El receptor descarta únicamente la trama errónea y acepta las que llegan detrás almacenándolas en el buffer de recepción. En esta situación falta una trama en la secuencia (tramas desordenadas). Al detectar la trama errónea envía una señal SREJ n (señal propia para este tipo de estrategia, n=nº trama errónea), o NACK n, (trama n no validada, n=nº trama errónea), al emisor para indicarle la situación. procediendo el emisor a reenviarle únicamente esta trama

errónea. Esta trama errónea se encontraría en el buffer del emisor a la espera de ser validada. El receptor al recibir la retransmisión correcta de la anterior trama errónea la almacena en el buffer con el resto de tramas recibidas y las ordena, para posteriores tratamientos. Con ello ha recibido la secuencia de tramas en orden correcto y válido.

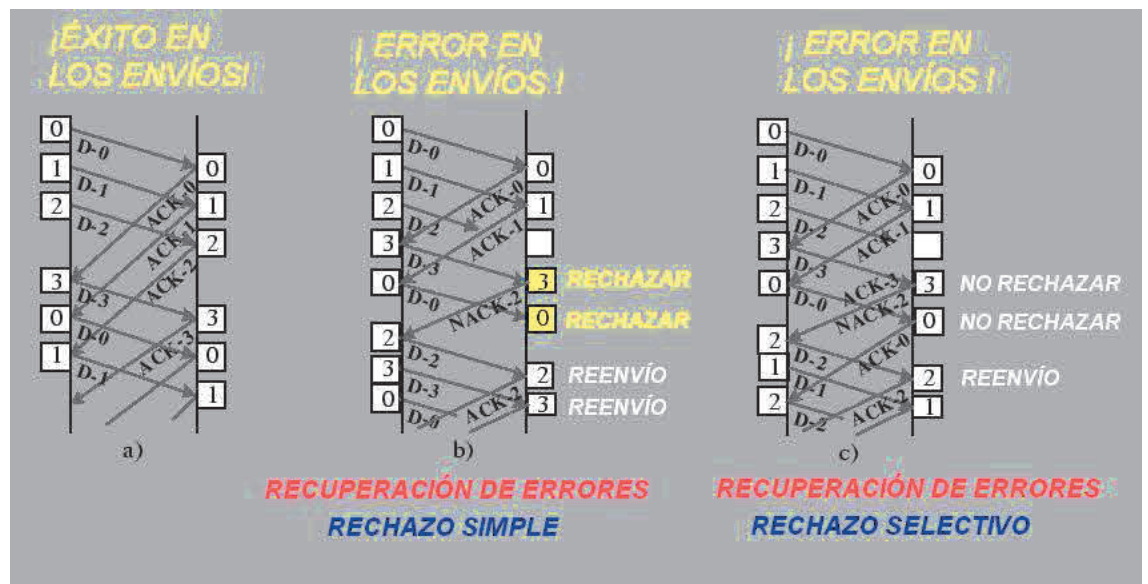


Ilustración 35: Funcionamiento ventana deslizante con recuperación de errores. Fuente: Wikipedia

4.4 Integración UART-CAN-Ventana deslizante

Para el desarrollo de este TFG y llevar a cabo esta integración vamos a partir del siguiente esquema, desarrollado en el entorno Fritzing:

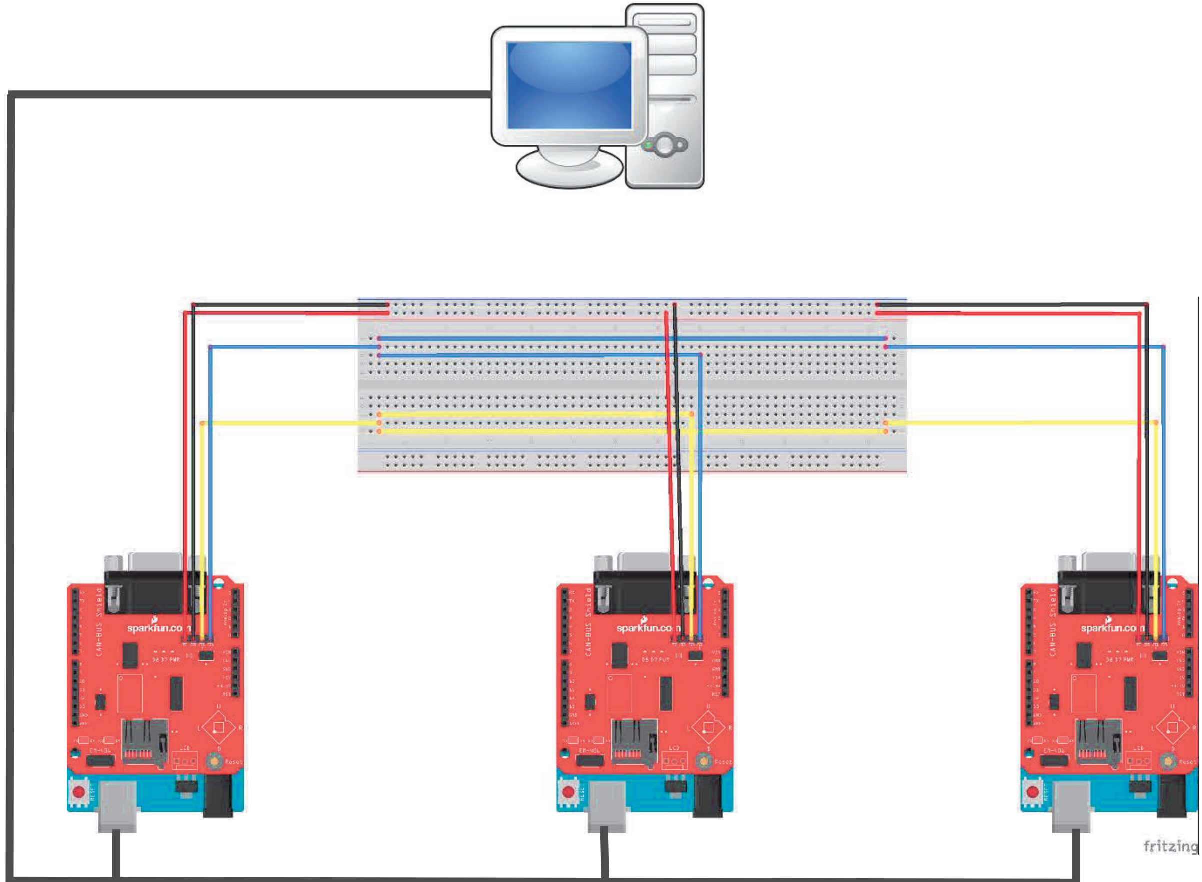


Ilustración 36: Esquema integración UART-CAN-Ventana_deslizante

Siguiendo esta distribución cada dispositivo tendrá el siguiente funcionamiento:

El PC será el encargado de enviar datos a través del monitor serie a cada uno de los dispositivos. Estos dispositivos conectados al bus CAN poseerán de izquierda a derecha los identificadores 1,2,3 respectivamente. Estos dispositivos tendrán 1 ventana tanto de emisión como de recepción, para el tráfico de tramas CAN, y obteniendo los datos serie enviados desde el PC y metiéndolos en las tramas CAN.

Para llevar a cabo todo lo expuesto, se desarrolló la librería SlidWind y Timer. La librería SlidWind cuyas funciones implementan el protocolo

de ventana deslizante, hace uso de la librería ArCan, anteriormente explicada, y la librería Timer. Estas dos librerías (Timer y SlidWind) se explicarán a continuación.

4.4.1 Librería Timer

El motivo del desarrollo de la librería Timer, fue para poder tener los mecanismos de temporización de las ventanas de emisión y recepción en la librería SlidWind.

Toda la funcionalidad de Timer es accesible mediante la clase tTimer y todas sus funciones son públicas, con lo que son accesibles por el usuario. Dicha clase tTimer tiene un parámetro llamado timer_ms_count inicializado a 0.

La librería Timer posee las siguientes fuciones:

void init_timer():

En esta función se inicia el Timer2 del Arduino UNO en el modo CTC, con un prescaler de 64, y en el registro OCR2A se carga el valor 249 de manera que dicho timer producirá una interrupción cada milisegundo.

void start_timer():

En esta función se pondrá el valor de los milisegundos transcurridos desde que se inició el Timer2 en la variable timer_ms_count.

void reset_timer():

Esta función reinicializará el timer poniendo la variable timer_ms_count a 0.

uint64_t get_value_of_timer():

Esta función devolverá el valor en el que se inició el timer devolviendo la variable timer_ms_count.

uint64_t get_time_ms():

Esta función nos devolverá el número de milisegundos transcurridos desde que se inició el Timer2.

4.4.2 Librería SlidWind

La librería SlidWind es la encargada de realizar el protocolo de ventana deslizante a través de sus funciones. Esta librería está pensada para el manejo del tráfico de tramas CAN por lo que hace uso de la librería ArCan que tal y como vimos anteriormente maneja los datos empaquetándolos en tramas CAN, así como el envío y recepción de las mismas. También se hace uso de la librería Timer y de varias funciones de la librería Serial, ya que, tal y como se introdujo antes, la ventana deslizante hace uso de temporizadores en la ventana de emisión y recepción y el uso de la librería Serial es para obtener y mostrar los datos a través del monitor serie de los dispositivos.

Esta librería SlidWind está implementada siguiendo el protocolo de ventana deslizante con rechazo selectivo.

Toda la funcionalidad de SlidWind es accesible mediante la clase tSlidW. Tiene una parte pública la cual es accesible por el usuario a través de una aplicación creada en Arduino y una parte privada que no es accesible por el usuario, pero cuyas funciones son usadas por las de la parte pública.

En la declaración de la clase tSlidW tenemos varios registros, los cuales son los siguientes:

```
typedef struct{
    //in-use-> saber si la entrada esta siendo usada, retries->numero de reintentos de envio
    bool in_uses=false;
    int retries=0;

    tArCan::tCAN packet;

    //Manejo trama CAN
    uint16_t source_id;
    uint16_t destination_id;
    tArCan::packet_type type_of;
    uint8_t sequence;
    uint8_t sequence_ack;

}entry;
```

Ilustración 37: Registro entry

Este registro se corresponde con cada una de las entradas de la ventana tanto de emisión como de recepción en la que tenemos:

- **in_use:** Nos indica si dicha entrada está siendo usada o no.

- **Retries:** Nos indica el número de intentos que la trama ha sido reenviada.
- **tArCan::tCAN packet:** Trama CAN que se encuentra almacenada en la entrada.
- **Source_id:** Id del emisor de la trama CAN.
- **Destination_id:** Id del receptor de la trama CAN.
- **tArCan::packet_type type_of:** tipo de trama CAN (DATA, ACK, NACK).
- **Sequence:** Número de secuencia de la trama CAN.
- **Sequence_ack:** Número de secuencia de la trama CAN que voy a confirmar.

```
typedef struct{
    tSlidW::entry       out_bufs   [NR_BUFS];
    tTimer              timers_out [NR_BUFS];
    int lo_lim_send_winds=-1;
    int up_lim_send_winds=-1;
    int send_occupieds=0;
    uint8_t sequence_now=0; //Siguiente número de secuencia que tengo que enviar
    double timeout=1000;
    uint32_t retries=0; //Número total de reenvíos realizados
    uint32_t confirms=0; //Número total de paquetes conirmados
}sender;
```

Ilustración 38: Registro sender

En este registro tenemos lo necesario para usar la ventana deslizante del emisor:

- **tSlidw::entry out_bufs [NR_BUFS]:** Buffer del emisor de tamaño NR_BUFS(=4).
- **tTimer timers_out[NR_BUFS]:** Temporizadores de cada una de las entradas del buffer del emisor.
- **lo_lim_send_winds:** Límite inferior de la ventana deslizante del emisor.
- **up_lim_send_winds:** Límite superior de la ventana deslizante del emisor.
- **send_occupieds:** Número de entradas ocupadas en el buffer del emisor.
- **sequence_now:** Número de secuencia actual de la trama.
- **Timeout:** Tiempo en milisegundos en el que se realizan los reenvíos
- **Retries:** Número de reenvíos realizados

- **Confirms:** Número de confirmaciones recibidas

```
typedef struct{
    tSlidW::entry        input_bufs [NR_BUFS];
    tTimer               timer_ack;
    int lo_lim_rec_winds=0;
    int up_lim_rec_winds=3;
    int rec_occupieds=0;
    int next_confirm=-1; //Confirmación que tengo que enviar al emisor
    double timeout=100;
    uint32_t goods_r=0; //Paquetes recibidos con las sequencias esperadas
    uint32_t bads_r=0;  //Paquetes recibidos fuera de las secuencias esperadas
}recipient;
```

Ilustración 39: Registro recipient

En este registro tenemos lo necesario para usar la ventana deslizante del receptor:

- **tSlidw::entry input_bufs [NR_BUFS]:** Buffer del receptor de tamaño NR_BUFS(=4).
- **tTimer timer_ack:** Temporizador para saber si hay que enviar un ack individual.
- **lo_lim_rec_winds:** Límite inferior de la ventana deslizante del receptor.
- **up_lim_rec_winds:** Límite superior de la ventana deslizante del receptor.
- **rec_occupieds:** Número de entradas ocupadas en el buffer del receptor.
- **Next_confirm:** Número de la secuencia de la trama que hay que confirmar.
- **Timeout:** Tiempo en que se obtienen mensajes del bus en milisegundos.
- **goods_r:** Número de paquetes recibidos cuyo número de secuencia se encuentra dentro de los límites de la ventana deslizante del receptor.
- **Bads_r:** Número de paquetes recibidos cuyo número de secuencia no se encuentra dentro de los límites de la ventana deslizante del receptor.

```
typedef struct{
    //Manejo TIMEOUT
    tTimer timer_timeout;
    //Transmisión
    uint32_t trans_tim=0;
    uint32_t trans_typi_dev=0;
    double trans_ave=0;
    double trans_dev=0;
    uint32_t trans_num_samples=0;
    uint8_t trans_sequence;
    bool trans;
    //Transferencia
    uint32_t transf_tim=0;
    uint32_t transf_typi_dev=0;
    double transf_ave=0;
    double transf_dev=0;
    uint32_t transf_num_samples=0;
}timeout;
```

Ilustración 40: Registro timeout

En este registro tenemos las variables necesarias para medir los tiempos de transmisión (tiempo en el que el MCP2515 envía una trama al MCP2551 para que la deposite en el bus CAN) y de transferencia (tiempo en el que el dispositivo envía una trama y le confirman que ha llegado correctamente), así como el cálculo de las medias y las desviaciones típicas de cada uno de estos tiempos. El cálculo de la media del tiempo de transferencia será clave ya que este factor podrá actualizar el timeout del emisor, que es el encargado de realizar los reenvíos cada cierto número de milisegundos. Esto se debe a que si una trama tarda X tiempo en ser confirmada para el dispositivo, este valor X ha de tenerse en cuenta para los reenvíos y así no saturar el bus con demasiados reenvíos.

A continuación, se presentan todas las funciones pertenecientes de la parte publica:

void init_sliding_window(uint32_t filter, uint32_t mask_ext, uint32_t mask_std):

Esta función es la encargada de inicializar el monitor serial, arrancar el Timer 2 y el bus CAN pasándole los parámetros filter, mask_ext y mask_std a la función init de la librería ArCan.

bool check_input_timers(recipient& rec, sender& sent):

Esta función es la encargada de comprobar si el timer de recepción (timer_ack) ha expirado. En caso de que haya expirado se manda una trama tipo ack que no contendrá datos (rtr=1), en el que introduciremos en sequence_ack del campo de arbitraje de la trama CAN el valor que posea en ese momento next_confirm, ya que esta variable está actualizada conteniendo el valor de la trama o tramas que me han llegado y por tanto las que se tienen que confirmar. Luego reiniciamos dicho timer.

bool check_output_timers(recipient& rec, sender& sent):

Aquí se comprueba que algún timer de envío haya expirado. En caso afirmativo quiere decir que no se ha recibido confirmación de esa trama cono lo que puede no haber llegado al destino, con lo que, se reenvía y reinicia el valor de su timer. También se reinicia el valor del timer de recepción timer_ack ya que en este reenvío se pueden estar confirmando tramas (tramas ACK) y no es necesario enviar un ack individual.

bool check_window_status(int send_occupieds, recipient rec):

Nos devuelve verdadero si la ventana tiene entradas libres en el emisor o en el receptor. Falso en caso contrario.

bool sent_packet(uint16_t id_source, uint16_t id_destination, uint8_t length, byte data[8], sender& sent, recipient& rec):

Esta función coge los datos que le pasamos por el vector data, la longitud length que será la longitud de estos datos, el id_source que será el id del emisor (generalmente el dispositivo que llame a esta función), id_destination que será el id del destinatario, construyendo una trama CAN con estos parámetros. Luego cogemos esta trama CAN y la metemos siguiendo un orden según su número de secuencia de la trama en el buffer del emisor.

Posteriormente mandamos esta trama por el bus CAN e incrementamos el número de entradas ocupadas y el límite superior de la ventana del emisor. Tras este envío iniciamos su respectivo timer así

como el timer de recepción timer_ack, ya que en esta trama enviada puedo estar confirmando alguna, también incrementamos la secuencia de cara al envío del siguiente paquete

bool receive_packet(recipient& rec, sender& sent):

En esta función se va tramas CAN durante un tiempo determinado. Este tiempo lo establece el usuario a través de la variable timeout del registro recipient.

Se obtiene el mensaje y se mira de que tipo es:

- Si es de tipo NACK quiere decir que se tiene que enviar la trama del buffer del emisor con mismo número de secuencia que el campo sequence_ack de la trama que llegó. Este proceso se realiza siempre y cuando el valor del campo sequence_ack se encuentre entre los límites de la ventana del emisor.
- Si es de tipo ACK se llama a la función check_ack que se explicará posteriormente.
- Si es de tipo DATA se mete en el buffer de recepción comprobando que la secuencia que llegó se encuentra entre las esperadas, en orden siguiendo el mismo esquema que el buffer de emisión. Una vez metida en el buffer se incrementa el número de entradas ocupadas, el límite superior de la ventana de emisión. Luego se incrementa el next_confirm según el número de secuencia que haya llegado en la trama para saber que tramas de las que han llegado correctamente confirmo.

void print_buffer (entry buffer [NR_BUFS]):

Nos muestra las tramas CAN contenidas dentro del buffer, comprobando las entradas que están siendo ocupadas que son sólo las que nos interesa.

void print_sender(sender sent):

Nos muestra todo el contenido del emisor, desde sus tramas CAN almacenadas dentro del buffer hasta los límites de su ventana y su próxima secuencia a enviar.

void print_recipient(recipient rec):

Nos muestra todo el contenido del receptor, desde sus tramas CAN almacenadas dentro del buffer hasta los límites de su ventana y su próxima secuencia a enviar.

bool get_data(uint16_t& id, uint8_t length, byte data[8], recipient& rec):

En este método obtenemos el identificador del emisor, los datos y la longitud de los mismo de la ventana del receptor. Posteriormente liberamos dicha entrada, incrementamos el límite inferior de la ventana del receptor y disminuimos las entradas ocupadas del receptor. En general, este método hay que usarlo para liberar entradas en el buffer de recepción, y mostrar los datos ordenadamente.

bool introduce_data(String& incom):

Este método es el encargado de recibir los datos introducidos por el usuario a través del monitor serie del dispositivo y almacena dichos datos en la variable incom. Devolverá verdadero si obtuvo datos y falso en caso contrario.

void get_byte_data(String& incom, byte data [8],uint8_t& length):

Este método es el encargado de ir cogiendo los datos de la variable incom, e ir almacenándolos en el array data, una vez almacenados en este array, estos datos son eliminados de la variable incom. El número de datos introducidos en el array data estará indicado por la variable length.

void print_timeout(timeout tim):

Este método es el encargado de mostrarnos por el monitor serie la media y la desviación típica de los tiempos de transmisión y transferencia obtenidos.

A continuación, se presentan todas las funciones de la parte privada, no accesibles por el usuario, pero de igual importancia para el funcionamiento de la librería ya que las funciones anteriormente de la parte pública hacen uso de estas.

void send_ack_message_wind(recipient& rec):

Este método se encarga de enviar una trama de confirmación individual (ACK) en nuestra ventana deslizante. Esto se consigue construyendo la trama CAN de forma que en el campo de arbitraje indicamos los id del emisor y del receptor, el tipo será tArCan::ack, el campo de secuencia lo pondremos a 0, y en el campo de confirmación de secuencia sequence_ack introduciremos el valor de la variable next_confirm del receptor que contendrá el valor actual de la trama que tengo que confirmar.

void send_nack_message_wind(sender& sent):

Este método se encarga de enviar una trama de no confirmación individual (NACK) en nuestra ventana deslizante. Esto se consigue construyendo la trama CAN de forma que en el campo de arbitraje indicamos los id del emisor y del receptor, el tipo será tArCan::nack, el campo de secuencia lo pondremos a 0, y en el campo de confirmación de secuencia al valor de la secuencia de la trama CAN que no hemos recibido.

bool send_message_wind(tArCan::tCAN message):

Esta función es la encargada de enviar tramas de datos, almacenándolas en el buffer de la ventana deslizante del emisor, devolviendo verdadero si se envió correctamente, y falso en caso contrario.

tArCan::tCAN construct_packet(uint16_t id_source, uint16_t id_destination, uint8_t length, byte data[8], uint8_t sequence_now, uint8_t next_confirm):

Este método nos construirá la trama CAN de datos metiendo en el campo de arbitraje las variables pasadas por parámetro id_source, id_destination, sequence=sequence_now, sequence_ack=next_confirm.

Luego la longitud de los datos de la trama será la variable length y el campo de datos contendrá lo que se encuentre en el vector data.

boolean between (uint8_t lo_lim, uint8_t up_lim, uin_8_t sequence):

Función que nos devolverá verdadero si el valor de sequence se encuentra entre lo_lim y up_lim y falso en caso contrario.

void copy_data(entry buffer, byte data[8]):

Método que copiará los datos contenidos en el vector data en el campo de datos de la trama CAN almacenada en la variable buffer.

void check_ack(sender& sent, uint8_t sequence_acks):

Este método nos irá confirmando tramas CAN de la ventana del emisor cuyo valor de secuencia sea menor o igual al valor de sequence_acks. Cuando nos referimos a confirmar, se trata de ir liberando las entradas del emisor que cumplan la condición anterior. A medida que se van liberando entradas el límite inferior de la ventana del emisor se va incrementando.

Void calc_ave_typidev(timeout& tim):

Este método será el encargado de realizar el cálculo de la media y la desviación típica de los tiempos de transmisión y transferencia.

4.5 Pruebas realizadas

Para realizar pruebas de este sistema se decidió realizarlas mediante la siguiente distribución:

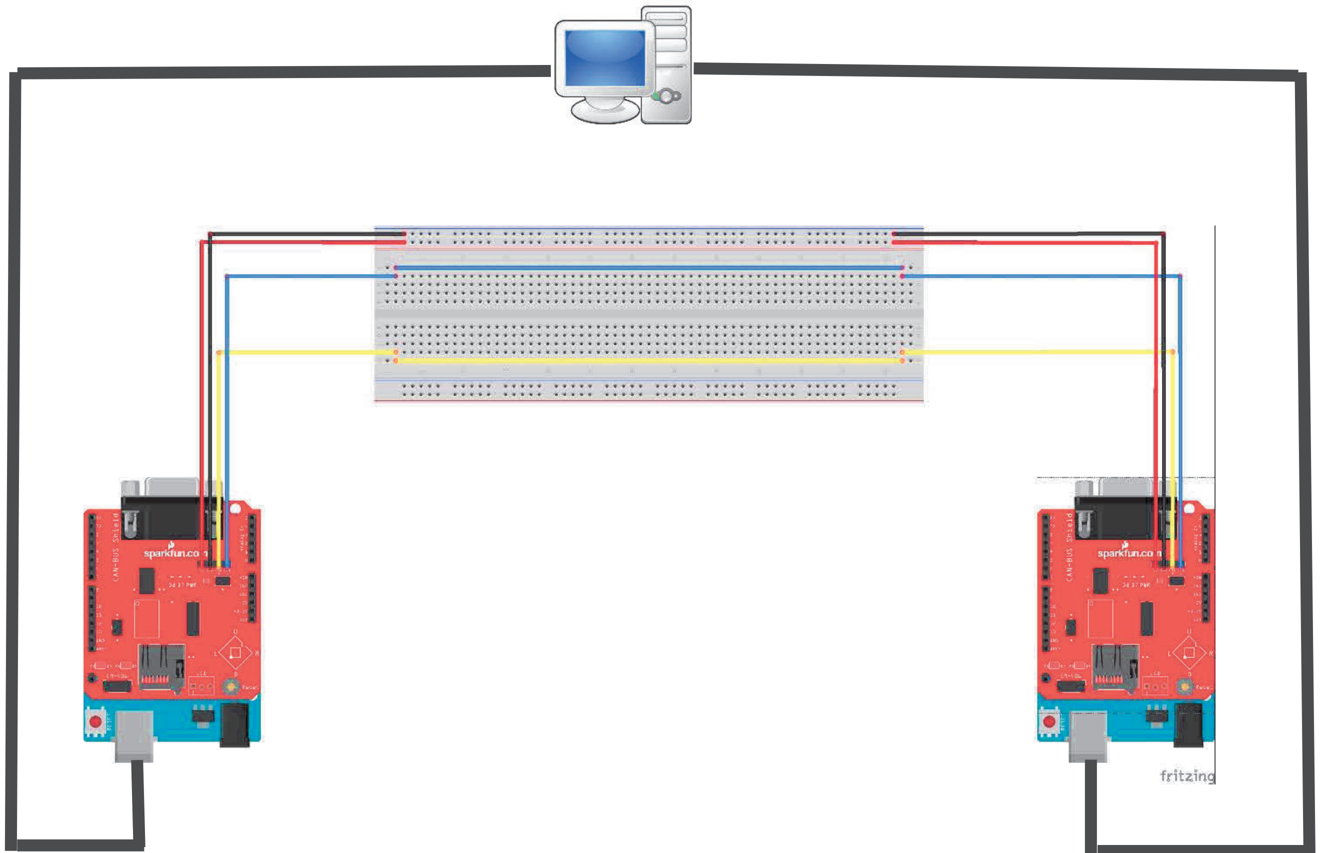


Ilustración 41: Esquema UART-CAN-VENTANA_DESLIZANTE para la realización de pruebas

En esta distribución elegida para las pruebas el funcionamiento será de la siguiente manera, el dispositivo de la izquierda tendrá identificador 1 y será el encargado de enviar tramas de datos, de reenviarlas cada cierto número de milisegundos, y de recibir las confirmaciones de las tramas de datos enviadas previamente. El dispositivo de la derecha poseerá identificador 2, tendrá cargado el ejemplo proporcionado "rate_siliding_rec.ino", y será el encargado de recibir las tramas de datos del dispositivo con identificador 1, teniendo cargado el ejemplo proporcionado "rate_sliding_sent.ino", y de enviar confirmaciones individuales ya que este dispositivo no enviará tramas de datos, estas confirmaciones individuales los enviará cada cierto número de milisegundos.

Se realizaron 10 ejecuciones con esta distribución de cada una de las distintas combinaciones disponibles cambiando el factor multiplicativo de los tiempos de reenvío en la emisión de tramas de datos (sent.timeout) del dispositivo con identificador 1, y cambiando el factor multiplicativo en el tiempo de envío de confirmación individual (rec.timeout) en el dispositivo con identificador 2. El valor inicial que posee el parámetro rec.timeout es de 100 milisegundos y el valor que posee el parámetro inicialmente es de 1000 milisegundos pudiendo actualizarse este parámetro, de manera que, si el cálculo de la media del tiempo de transferencia obtenido es menor que este valor, sent.timeout pasará a tener el valor del tiempo de transferencia, siendo este parámetro modificado en la ejecución.

En cada una de las ejecuciones se trataba de enviar 1000 tramas de datos desde el dispositivo 1 y el dispositivo 2 debía recibir las 1000 tramas de datos y enviar al dispositivo 1 ack´s para confirmar estas tramas recibidas.

Estos factores multiplicativos para los tiempos de reenvío de tramas de datos y para los tiempos de envío de confirmaciones individuales podrán tener valores 2,3,4 ya que son los casos más interesantes ya que en el caso del emisor (sent.timeout) si establecemos este factor multiplicativo <= 4 no se producen reenvíos de trama y en el caso del receptor (rec.timeout) para un valor mayor que 3 ya los tiempos de envío de confirmación serán muy grandes.

Los valores obtenidos en cada una de las ejecuciones se tratan de la media de los tiempos de transmisión y transferencia obtenidos a lo largo de la ejecución del programa.

Como resultados a los tiempos de transmisión (tiempo en el que el MCP2515 entrega la trama al MCP2551 para que la envíe por el bus) se obtuvieron los siguientes datos:

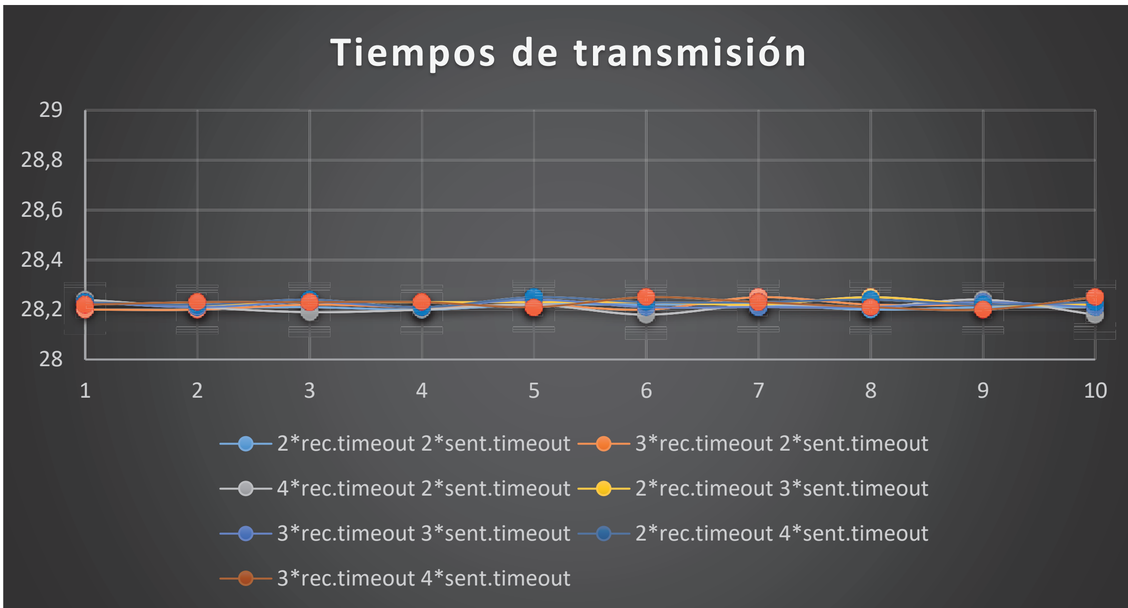


Ilustración 42: Gráfica tiempos de transmisión

Como podemos ver en la gráfica los tiempos de transmisión no varían mucho entre los distintos casos ya que como máximo se obtienen valores de 28,25 milisegundos y como mínimo 28,18 milisegundos.

Con respecto a los tiempos de transferencia (tiempo transcurrido desde que se envía una trama y se recibe la confirmación de la misma) se obtuvieron los siguientes valores:

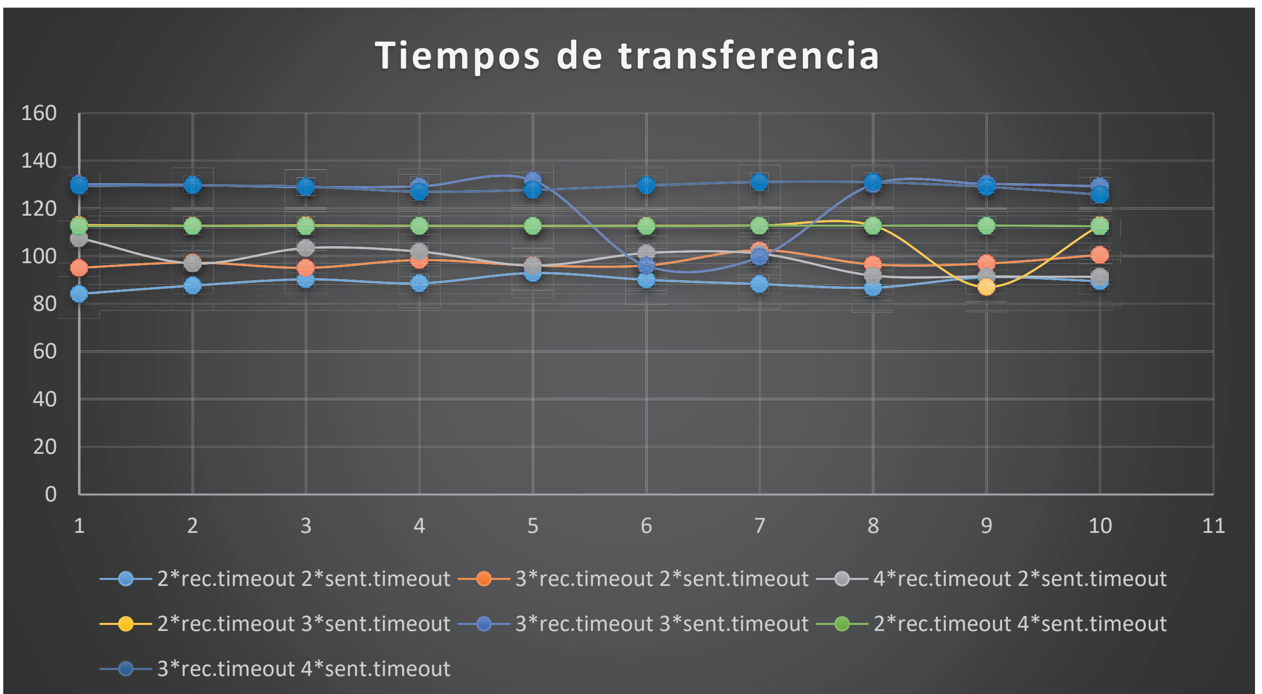


Ilustración 43: Gráfica tiempos de transferencia

Como se observa en los resultados obtenidos el caso en el que se obtienen peores tiempos de transferencia es “3*rec.timeout 4*sent.timeout” esto se debe a que al tardarse mucho en enviar las confirmaciones individuales y reenviar las tramas, suponiendo que sea necesario hacerlo, el tiempo de transferencia aumentará.

El mejor tiempo de transferencia obtenido es el caso “2*rec.timeout 2*sent.timeout”, esto nos verifica lo explicado anteriormente ya que al tener factores multiplicativos bajos se obtendrán tiempos menores, y por tanto, tiempos de reenvío de confirmaciones más cortos, incluso confirmándose varios envíos en una sola confirmación.

En este caso los tiempos de transferencia obtenidos son de aproximadamente unos 80 milisegundos. Obteniendo una frecuencia de 1/0.080=12.5 Hz, pudiendo decir que en cada segundo puedo confirmar unas 12,5 tramas de datos. Si cada trama de datos que se envía por el bus tiene 16 bytes al usarse el formato de tramas CAN extendido. El ancho de banda del canal será unos 1.6 kbps. Teniendo en cuenta que este shield está capacitado con la versión 2.0B del bus CAN, pudiendo llegar a 1 MB/s de ancho de banda, se podría decir que nuestro canal de comunicaciones no es eficiente. Este resultado se debe principalmente a que el sentido de las pruebas realizadas fue el de verificar el comportamiento del algoritmo de la ventana deslizante cuyo tiempo viene afectado por los tiempos que lo parametrizan: el tiempo de envío de confirmaciones individuales, tiempo en que se producen los reenvíos, tiempo de espera de recepción de mensajes, etc. A lo que habría que añadir el tiempo en esta batería de pruebas en las que se muestra información por el monitor serie que ralentiza enormemente el tiempo de transmisión medido. Por otro lado, el propio diseño del flujo de recepción y envío entre receptor y emisor en las pruebas realizadas influencia enormemente en los tiempos medidos.

A continuación, se presenta el número de tramas de datos enviadas, confirmaciones recibidas y tramas de datos reenviadas para el receptor:

	Paquetes enviados	Paquetes Reeenviados	Confirmaciones Recibidas
2*rec.timeout 2*sent.timeout	1000	360	999
2*rec.timeout 3*sent.timeout	1000	0	999
2*rec.timeout 4*sent.timeout	1000	0	1000
3*rec.timeout 2*sent.timeout	1000	421	997
3*rec.timeout 3*sent.timeout	1000	72	999
3*rec.timeout 4*sent.timeout	1000	71	1000
4*rec.timeout 2*sent.timeout	1000	678	989

El caso en el que se obtiene mayor tráfico es en el caso “4*rec.timeout 2*sent.timeout” esto se debe a que se tarda mucho en enviar las confirmaciones pero los reenvíos se producen muy rápidamente produciendo una saturación de la red CAN considerable ya que se necesitan muchos reenvíos para confirmar la trama.

Los mejores casos en los que apenas se producen reenvíos son “2*rec.timeout 3*sent.timeout” “2*rec.timeout 4*sent.timeout” esto se produce a que al realizarse los reenvíos en un intervalo grande de tiempo y los envíos de las confirmaciones individuales se realizan en un plazo corto de tiempo, no hay necesidad de reenviar dicha trama de datos ya que se confirma antes de que se vaya a producir el reenvío.

En el receptor se contabilizó los paquetes que se recibieron “bien” o en secuencia (paquetes recibidos que se encuentran dentro de la ventana

deslizante) y los paquetes que se recibieron "mal" o fuera de secuencia (paquetes recibidos que no se encuentran dentro de la ventana deslizante):

	Paquetes bien recibidos	**Paquetes mal recibidos**
2*rec.timeout 2*sent.timeout	1000	270
2*rec.timeout 3*sent.timeout	1000	0
2*rec.timeout 4*sent.timeout	1000	0
3*rec.timeout 2*sent.timeout	1000	327
3*rec.timeout 3*sent.timeout	1000	72
3*rec.timeout 4*sent.timeout	1000	71
4*rec.timeout 2*sent.timeout	1000	671

Aquí como vemos sucede lo mismo que en el tráfico del emisor como era de esperar, ya que, el caso en el que se obtiene mayor tráfico es en el caso "4*rec.timeout 2*sent.timeout".Ésto se debe a que se tarda mucho en enviar las confirmaciones, pero los reenvíos se producen muy rápidamente. Los mejores casos son "2*rec.timeout 3*sent.timeout" "2*rec.timeout 4*sent.timeout" ello se produce a que al realizarse los reenvíos en un intervalo grande de tiempo y los envíos de las confirmaciones individuales se realizan en un plazo corto de tiempo.

5. Conclusiones

Durante el desarrollo de este TFG se han logrado los siguientes aspectos:

1. Análisis y estudio del protocolo Controller Area Network(CAN), realizando un estudio exhaustivo de la versión 2.0B.
2. Análisis y estudio del MCP2515 y del MCP2551, para el entendimiento de su uso en la librería ArCan.
3. Análisis, estudio y mejora de la librería ArCan, corrigiendo ciertos errores que poseía en la parte del uso de las tramas extendidas y añadiéndole nuevas funcionalidades como el uso de modo dormido, tipo de paquete (tramas ACK, NACK y de Datos), la impresión de información de depuración de los distintos tipos de trama.
4. Diseño desarrollo y prueba de una librería de ventana deslizante. Concretamente usando un mecanismo de ventana deslizante de repetición selectiva en el que solo se producen reenvíos de trama de las tramas no confirmadas por el receptor. También se desarrolló una pequeña librería para el uso y manejo del Timer 2 de Arduino UNO, necesaria para los temporizadores usados en el mecanismo de ventana deslizante.
5. Desarrollo de varios ejemplos localizados en la carpeta Examples en la que podemos encontrar un ejemplo para el uso de la ventana deslizante, un ejemplo para la medición de tiempos que fue el usado para la batería de pruebas realizadas. Así como un ejemplo de comunicación entre dos dispositivos a través de bus CAN que mapean un puerto serie entre ellos, objetivo principal de este TFG.

Como se ha podido observar en el transcurso de este documento se ha desarrollado una red CAN bidireccional, transformando la información obtenida de dispositivos UART [14] en tramas CAN. Esto nos permite tener todos los dispositivos de nuestra red, que están conectados al bus, identificados gracias a los identificadores que usa el bus CAN los cuales actúan de filtro de mensajes extrayendo del bus sólo los mensajes que van dirigidos a sí mismos. También como optimización de pines con el

uso de esta integración se ha producido una reducción considerable del uso de los pines de nuestros dispositivos Arduino, los cuales al estar libres pueden ser usados ahora para otras funciones. Esta optimización de pines para interconectar varios dispositivos UART fue la idea de partida para el desarrollo de este TFG.

Al añadirse un protocolo de ventana deslizante con repetición selectiva, añadimos a nuestra red un mecanismo de control de flujo de datos entre el emisor y el receptor, garantizando que toda la información enviada va a ser recibida gracias a la repetición selectiva. Por otro lado, el objetivo principal del uso de este algoritmo de flujo de control es el de maximizar el uso del canal. Al ser este protocolo de comunicación bidireccional, el nodo actuará a veces de emisor y otras de receptor, tal y como se observa en el funcionamiento del bus CAN.

6. Trabajos Futuros

Como trabajos futuros a este desarrollo se pueden plantear las siguientes ideas:

- Desarrollo de un nodo centralizador que será el encargado de establecer la comunicación con el resto de dispositivos conectados al bus CAN. Este nodo centralizador será el único que estará comunicado con el PC. A su vez, el resto de los nodos conectados al bus serán los que obtengan datos de los dispositivos serie haciendo la funcionalidad desarrollada en este TFG. Una posible distribución podría ser la siguiente:

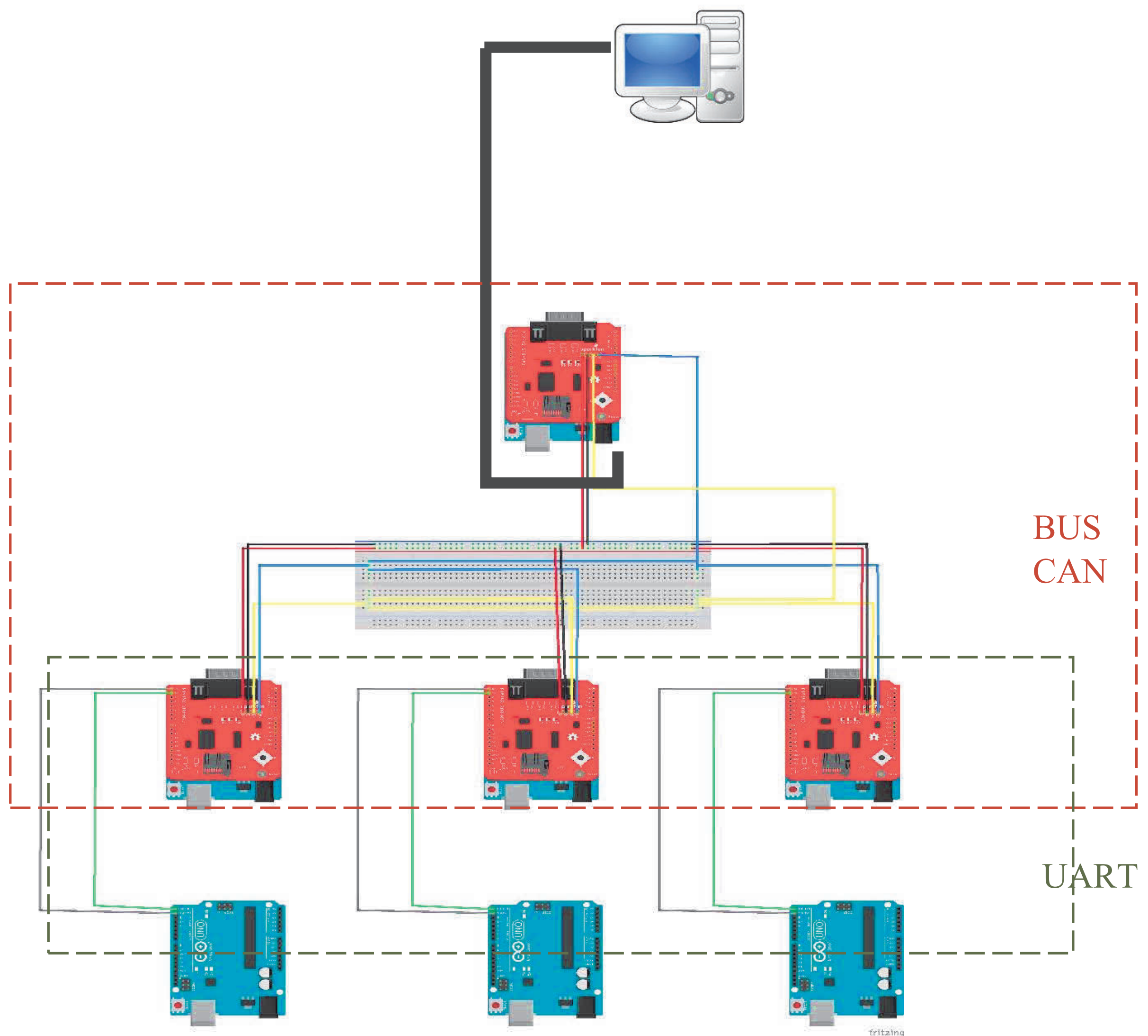


Ilustración 45: Esquema integración UART-CAN-Ventana_deslizante con nodo centralizador

- Añadir el estándar OpenCan.
- Mejora de la librería SlidWind desarrollada, realizando optimizaciones de su código y en la medida de lo posible añadirle nuevas funcionalidades.
- Diseñar una placa que haga directamente el intercambio de protocolo de UART a bus CAN, partiendo del uso de los microcontroladores MCP2515 [5] y MCP2551 [6] encargados del manejo del bus CAN.

7. Agradecimientos

Agradecer a mis tutores Antonio Carlos Domínguez Brito y Jorge Cabrera Gámez por su asesoramiento y apoyo durante el desarrollo de este Trabajo Fin de Grado, y a mi familia y amigos por apoyarme durante todos estos años de estudio y sacrificio.

Muchas gracias.

8.Referencias

[1] Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete. [En línea] Available:

http://www.atmel.com/images/atmel-42735-8-bit-avr-microcontroller-atmega328-328p_datasheet.pdf

[2] Understanding and Using the Controller Area Network Communication Protocol (Theory and Practice). Marco Di Natale, Haibo Zeng, Paolo Giusto, Arkadeb Ghosal. Springer.

[3] Wikipedia <<Bus CAN>> [En línea]. Available: https://es.wikipedia.org/wiki/Bus_CAN

[4] GII-SETR-Controller Area Network (CAN). Antonio Carlos Domínguez Brito.

[5] Datasheet MCP2515 [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21801e.pdf

[6] Datasheet MCP2551 [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21667f.pdf

[7] Cuarta Edición. Redes de Computadores. Andrew S. Tanenbaum. Pearson Prentice Hall.

[8] Wikipedia <<Universal Asynchronous Receiver-Transmitter>>[En línea]. Available:

https://es.wikipedia.org/wiki/Universal_Asynchronous_Receiver-Transmitter

[9] SIANI, «División de Robótica y Oceanografía Computacional,» [En línea]. Available:

http://www.roc.siani.es/

[10] Arduino Uno [En línea]. Available:

https://store.arduino.cc/arduino-uno-rev3

[11] CANBUS Shield Sparkfun [En línea]. Available:

https://www.sparkfun.com/products/13262

[12] Librería ArCan [En línea]. Available:

http://www.arcan.es/?page_id=394

[13] Arduino IDE [En línea]. Available:

https://www.arduino.cc/en/main/software

[14] Librería monitor Serial de Arduino [En línea].Available:

https://www.arduino.cc/reference/en/language/functions/communication/serial/

[15] ISO 11898[En línea]. Available:

https://www.iso.org/standard/63648.html

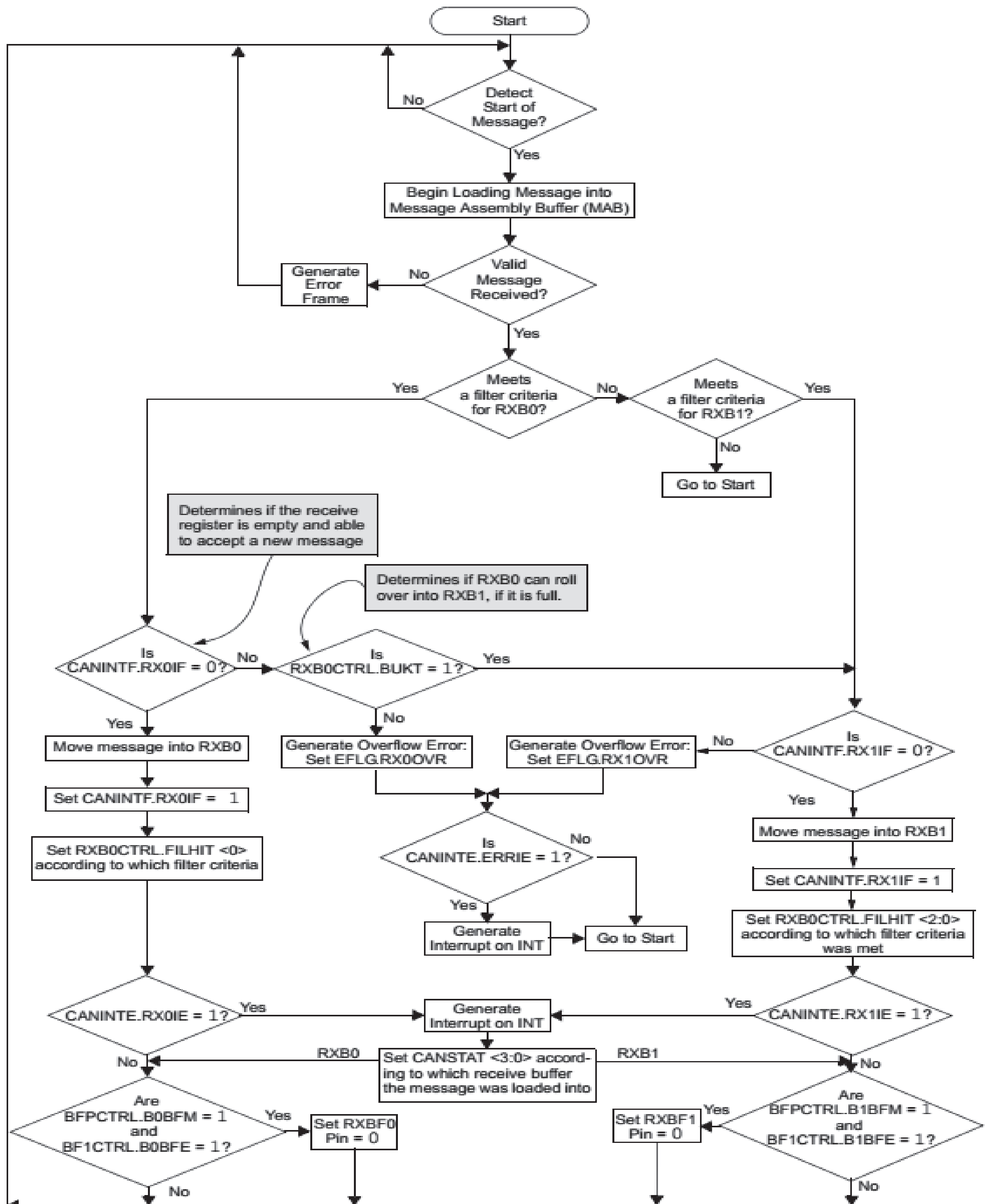
[16] Universidad de Málaga [En línea]. Available:

https://www.uma.es/

9.Anexo

9.1. Anexo 1

En este anexo se adjunta el diagrama de flujo sobre cómo se realiza la recepción de mensajes.



9.2. Anexo 2

En este anexo se adjunta el diagrama de flujo sobre cómo se realiza la transmisión de mensajes.