

# ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



## TRABAJO FIN DE GRADO

### DESARROLLO DEL FIRMWARE DE CONTROL DE LOS NODOS DE UNA PLATAFORMA EXPERIMENTAL DE WSN

**Titulación:** Grado en Ingeniería en Tecnologías de la Telecomunicación

**Mención:** Sistemas Electrónicos

**Autor:** Ricardo Antonio Rios Peña

**Tutores:** Dr. Roberto Esper-Chaín Falcón

Dr. Félix B. Tobajas Guerrero

**Fecha:** Julio 2015



## ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



### TRABAJO FIN DE GRADO

### DESARROLLO DEL FIRMWARE DE CONTROL DE LOS NODOS DE UNA PLATAFORMA EXPERIMENTAL DE WSN

### HOJA DE FIRMAS

**Alumno**

Fdo.: Ricardo Antonio Rios Peña

**Tutor**

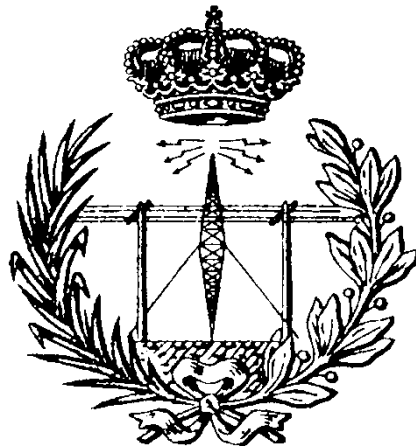
**Tutor**

Fdo.: Dr. Roberto Esper-Chaín Falcón Fdo.: Dr. Félix B. Tobajas Guerrero

**Fecha: Julio 2015**



# ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



## TRABAJO FIN DE GRADO

### DESARROLLO DEL FIRMWARE DE CONTROL DE LOS NODOS DE UNA PLATAFORMA EXPERIMENTAL DE WSN

### HOJA DE EVALUACIÓN

Calificación: \_\_\_\_\_

**Presidente**

Fdo.: Presidente

**Vocal**

Fdo.: Vocal

**Secretario/a**

Fdo.: Secretario

**Fecha: Julio 2015**



# Índice de Contenidos

CAPÍTULO 1: INTRODUCCIÓN .....	1
1.1 ANTECEDENTES .....	1
1.2 OBJETIVOS.....	5
1.3 SOLUCIÓN ADOPTADA.....	7
1.4 ESTRUCTURA DE LA MEMORIA. ....	8
CAPÍTULO 2: DESCRIPCIÓN HARDWARE .....	11
2.1 INTRODUCCIÓN. ....	11
2.2 MÓDULOS SENSORES INALÁMBRICOS. ....	11
2.3 MICROCONTROLADOR ATMEL AT90CAN. ....	15
2.3.1 Descripción general.....	15
2.3.2 La arquitectura AVR.....	17
2.3.3 Diagrama de bloques del microcontrolador AT90CAN128. ....	19
2.4 FUNCIONALIDADES DEL MICROCONTROLADOR AT90CAN128. ....	22
2.4.1 Puertos de entrada/salida de propósito general (GPIO). ....	23
2.4.1.1 Configuración del Pin. ....	25
2.4.2 USART.....	27
2.4.2.1 Generación de Reloj.....	29
2.4.2.2 Formato de la trama .....	31
2.4.2.3 Transmisor. ....	32
2.4.2.4 Receptor. ....	34
2.4.2.5 Descripción de registros.....	35
2.4.3 CAN.....	40
2.4.3.1 Protocolo CAN.....	41
2.4.3.2 Controlador CAN. ....	44
2.4.3.3 Objetos de mensajes (MOB). ....	45
2.4.3.4 Interrupciones.....	48
2.4.3.5 Descripción de los registros generales.....	48
2.4.3.6 Descripción de los registros MOB. ....	52
2.4.4 Bootloader.....	57
CAPÍTULO 3: DESARROLLO SOFTWARE .....	63
3.1 INTRODUCCIÓN. ....	63

3.2	DESCRIPCIÓN GENERAL .....	63
3.3	SISTEMA DE INTERRUPCIONES. ....	66
3.3.1	Vector de interrupción CAN. ....	66
3.3.2	Vector de interrupción USART: transmisión. ....	68
3.3.3	Vector de interrupción USART: recepción. ....	71
3.4	BIBLIOTECA CAN.....	74
3.4.1	Configuración previa. ....	75
3.4.2	Funciones públicas: configuración del bus. ....	75
3.4.3	Funciones públicas: transmisión. ....	76
3.4.4	Funciones públicas: recepción. ....	77
3.5	APLICACIÓN ROUTER. ....	78
3.6	APLICACIÓN BOOTLOADER. ....	83
CAPÍTULO 4: RESULTADOS.....		91
CAPÍTULO 5: CONCLUSIONES Y LÍNEAS FUTURAS .....		101
5.1	CONCLUSIONES. ....	101
5.2	LÍNEAS FUTURAS. ....	103
REFERENCIAS BIBLIOGRÁFICAS .....		105
PLIEGO DE CONDICIONES.....		107
PL.1	CONDICIONES HARDWARE.....	107
PL.2	CONDICIONES SOFTWARE.....	107
PRESUPUESTO .....		109
P.1	RECURSOS MATERIALES.....	109
P.1.1	Recursos Hardware. ....	109
P.1.2	Recursos Software. ....	110
P.2	TRABAJO TARIFADO POR TIEMPO EMPLEADO. ....	111
P.3	MATERIAL FUNGIBLE. ....	112
P.4	COSTES DE REDACCIÓN DEL TRABAJO FIN DE GRADO. ....	113
P.5	DERECHOS DE VISADO DEL COITT.....	114
P.6	COSTES DE TRAMITACIÓN Y ENVÍO.....	114
P.7	APLICACIÓN DE IMPUESTOS. ....	115
ANEXO I. CONTENIDO DEL CD-ROM .....		117
A.I.I	ESTRUCTURA DEL CD-ROM.....	117



# Índice de Figuras

FIGURA 1.1. WIRELESS SENSOR NETWORK.....	2
FIGURA 1.2. CONFIGURACIÓN EN BUS .....	5
FIGURA 1.3. ESTRUCTURA DE LA PLATAFORMA.....	7
FIGURA 2.1. PCB DE LOS MÓDULOS SENSORES INALÁMBRICOS.....	12
FIGURA 2.2. CHIP FT232R.....	13
FIGURA 2.3. ESQUEMÁTICO DEL CIRCUITO USB A UART .....	13
FIGURA 2.4. CHIP SN65HVD255 .....	14
FIGURA 2.5. CONECTOR CAN.....	14
FIGURA 2.6. ESQUEMÁTICO DEL CIRCUITO TRANSECTOR DEL CAN.....	15
FIGURA 2.7. ATMEL AT90CAN128 MICROCONTROLLER .....	17
FIGURA 2.8. DIAGRAMA DE BLOQUES DEL MICROCONTROLADOR AT90CAN128.....	20
FIGURA 2.9. ARQUITECTURA AVR.....	22
FIGURA 2.10. ESQUEMÁTICO EQUIVALENTE DE UN PIN I/O.....	23
FIGURA 2.11. I/O DIGITAL GENERAL .....	25
FIGURA 2.12. DESCRIPCIÓN DE REGISTROS DE LOS PUERTOS I/O.....	26
FIGURA 2.13. DIAGRAMA DE BLOQUES DE LA USART .....	28
FIGURA 2.14. LÓGICA DE GENERACIÓN DE LA SEÑAL DE RELOJ DE LA USART .....	30
FIGURA 2.15. ECUACIONES PARA CALCULAR LA CONFIGURACIÓN DE LA TASA DE BAUDIOS.....	31
FIGURA 2.16. FORMATO DE TRAMA.....	31
FIGURA 2.17. REGISTRO DE DATOS I/O DE LA USART .....	35
FIGURA 2.18. REGISTRO DE ESTADO Y DE CONTROL A.....	36
FIGURA 2.19. REGISTRO DE ESTADO Y DE CONTROL B.....	37
FIGURA 2.20. REGISTRO DE ESTADO Y DE CONTROL C.....	38
FIGURA 2.21. CONFIGURACIÓN DEL BIT UMSELN.....	38
FIGURA 2.22. CONFIGURACIÓN DE LOS BITS UPMN .....	39
FIGURA 2.23. CONFIGURACIÓN DEL BIT USBSN .....	39
FIGURA 2.24. CONFIGURACIÓN DE LOS BITS UCSZN .....	40
FIGURA 2.25. TRAMA CAN ESTÁNDAR.....	42
FIGURA 2.26. TRAMA CAN EXTENDIDA.....	43
FIGURA 2.27. ARBITRAJE DEL BUS .....	44
FIGURA 2.28. ESTRUCTURA DEL CONTROLADOR CAN .....	45

FIGURA 2.29. CONFIGURACIÓN DE LOS MOBS .....	46
FIGURA 2.30. REGISTRO DE CONTROL GENERAL .....	49
FIGURA 2.31. REGISTRO DE INTERRUPCIONES GENERALES .....	49
FIGURA 2.32. REGISTRO DE HABILITACIÓN DE INTERRUPCIONES GENERALES .....	50
FIGURA 2.33. REGISTROS DE HABILITACIÓN DE LOS MOBS.....	51
FIGURA 2.34. REGISTROS DE HABILITACIÓN DE LAS INTERRUPCIONES DE LOS MOBS.....	51
FIGURA 2.35. REGISTRO DE PÁGINA MOB.....	52
FIGURA 2.36. REGISTRO DE ESTADO DEL MOB .....	53
FIGURA 2.37. REGISTRO DE CONTROL Y DLC DEL MOB.....	53
FIGURA 2.38. REGISTRO DE IDENTIFICADOR .....	54
FIGURA 2.39. REGISTRO DE MÁSCARA DEL IDENTIFICADOR.....	56
FIGURA 2.40. EJEMPLO DE MÁSCARA .....	57
FIGURA 2.41. REGISTRO DE DATOS DEL MENSAJE .....	57
FIGURA 2.42. SECCIONES DE LA MEMORIA FLASH.....	59
FIGURA 2.43. PUNTERO Z .....	60
FIGURA 2.44. PÁGINAS DE LA MEMORIA FLASH .....	60
FIGURA 2.45. DIRECCIONAMIENTO DE LA MEMORIA FLASH DURANTE SPM .....	61
FIGURA 3.1. ESTRUCTURA DE LA PLATAFORMA .....	64
FIGURA 3.2. DIAGRAMA DE FLUJO DE LA PROGRAMACIÓN DE UN NODO.....	65
FIGURA 3.3. DIAGRAMA DE FLUJO DE LA RUTINA DE SERVICIO CAN .....	67
FIGURA 3.4. DIAGRAMA DE FLUJO DE LA FUNCIÓN INSERTXFIFO .....	69
FIGURA 3.5. DIAGRAMA DE FLUJO DE LA FUNCIÓN TxCHAR.....	70
FIGURA 3.6. DIAGRAMA DE FLUJO DE LA RUTINA DE SERVICIO USART: TRANSMISIÓN .....	71
FIGURA 3.7. DIAGRAMA DE FLUJO DE LA RUTINA DE SERVICIO USART: RECEPCIÓN .....	72
FIGURA 3.8. DIAGRAMA DE FLUJO DE LA FUNCIÓN TERMINATERX.....	73
FIGURA 3.9. DIAGRAMA DE FLUJO DE LA APLICACIÓN ROUTER.....	79
FIGURA 3.10. DIAGRAMA DE FLUJO DE LA FUNCIÓN PARSELINE.....	82
FIGURA 3.11. DIAGRAMA DE FLUJO DE LA APLICACIÓN BOOTLOADER .....	84
FIGURA 3.12. DIAGRAMA DE FLUJO DE LA FUNCIÓN PARSELINE.....	88
FIGURA 4.1. MONTAJE PRÁCTICO DE LA PLATAFORMA EXPERIMENTAL.....	91
FIGURA 4.2. INTERFAZ APLICACIÓN SERVIDOR .....	93
FIGURA 4.3. CONEXIÓN CON EL ROUTER.....	93
FIGURA 4.4. ESTABLECER DIRECCIÓN CAN DEL NODO .....	94
FIGURA 4.5. CARGAR CÓDIGO FUENTE .....	95
FIGURA 4.6. FICHERO .HEX GENERADO POR EL COMPILADOR .....	95

FIGURA 4.7. MENSAJES DE ESTADO DE LA APLICACIÓN SERVIDOR.....	96
FIGURA 4.8. EJECUTAR PROGRAMACIÓN DEL NODO .....	96
FIGURA 4.9. MENSAJE INICIAL DE LA APLICACIÓN BOOTLOADER.....	97
FIGURA 4.10. PORCENTAJE COMPLETADO DE LA PROGRAMACIÓN DEL NODO .....	97
FIGURA 4.11. MENSAJES DE CONFIRMACIÓN DEL NODO .....	98
FIGURA 4.12. LANZAR APLICACIÓN DE USUARIO .....	98
FIGURA 4.13. APLICACIÓN DE EJEMPLO EN EL NODO .....	99
FIGURA 4.14. MENSAJES RECIBIDOS EN EL ROUTER .....	99

## Índice de Tablas

TABLA PL.1. ELEMENTOS HARDWARE.....	107
TABLA PL.2. ELEMENTOS SOFTWARE .....	107
TABLA P.1. RECURSOS HARDWARE .....	110
TABLA P.2. RECURSOS SOFTWARE .....	110
TABLA P.3. FACTOR DE CORRECCIÓN SEGÚN LAS HORAS TRABAJADAS .....	111
TABLA P.4. COSTES MATERIAL FUNGIBLE.....	112
TABLA P.5. PRESUPUESTO ACUMULADO .....	113
TABLA P.6. COSTE TOTAL DEL TRABAJO FIN DE GRADO .....	115



# Resumen

En los últimos años, las redes de sensores inalámbricas han ganado mucha popularidad, experimentando un gran desarrollo y despliegue. La base de este auge proviene de maximizar la utilización de una de sus principales ventajas: la movilidad. Las redes de sensores inalámbricos presentan un gran potencial para aplicaciones en escenarios como el seguimiento y la vigilancia de objetivos militares, atenuación de desastres naturales, monitorización biomédica de la salud, exploración en entornos peligrosos y detección de seísmos, etc.

No obstante, este tipo de redes también presentan una serie de desventajas, siendo una de ellas el que, a pesar de que emplean tecnología inalámbrica, la programación y configuración de los nodos por lo general debe realizarse con el dispositivo físicamente conectado. También se hace muy difícil la depuración y el control de la red, ya que cuando se detecta que ocurre algún fallo en alguno de los nodos, la única forma que habría de comprobar qué es lo que está pasando, es desplazarse al lugar donde se encuentra dicho nodo y realizar una depuración *in situ*. Las desventajas que esto trae consigo son evidentes, ya que tener que ir de nodo en nodo verificando su funcionamiento puede representar un trabajo muy engorroso.

El objetivo principal del presente Trabajo Fin de Grado consiste en realizar una infraestructura que facilite la experimentación en el desarrollo de redes de sensores inalámbricos. La estructura estará conformada por una serie de nodos inalámbricos, los cuales estarán conectados a un bus CAN común, por un *router* y un servidor central. De esta forma, se dispondrá de un servidor desde el cual será posible programar todos los nodos conectados al bus, haciendo uso de un *router* que permitirá direccionar cada uno de ellos, además de poder recibir mensajes provenientes de dichos nodos. Para ello se desarrollará el firmware correspondiente a la aplicación *router* y a la aplicación *bootloader* de los nodos inalámbricos. Con ello se creará una infraestructura que permitirá programar y depurar los nodos de forma remota, concentrando los datos en un servidor central.



# Abstract

In recent years, wireless sensor networks have gained much popularity, experiencing a great development and deployment. The key to this growth comes from maximizing the use of one of its main advantages: mobility. Wireless sensor networks have great potential for applications in scenarios such as monitoring and surveillance of military targets, mitigation of natural disasters, biomedical health monitoring, exploration in hazardous environments, earthquakes detection, etc.

However, these kind of networks also have a number of disadvantages, one of which is that despite employing wireless technology, programming and configuration of the nodes generally has to be performed physically connected to the device. It is also very difficult to debug and control the network, because when a failure in one of the nodes is detected, the only way to check what is going on would be to go to the place where the node is located and start debugging it *in-situ*. The disadvantages this entails are obvious, as it might be a very cumbersome job to go from node to node checking their operation.

The main goal of this work is to develop an infrastructure that facilitates experimentation in the development of wireless sensor networks. The structure will consist of a series of wireless nodes which are connected to a common CAN bus, a router and a central server. Thereby a server will be available from which it will be possible to program all nodes connected to the bus, using a router that will address each of one of them, also the server can receive messages coming from these nodes. For this purpose the firmware corresponding to the router application and the wireless nodes bootloader application will be developed. This will create an infrastructure that will allow to program and debug nodes remotely, concentrating the data on a central server.





# Acrónimos

ACK	ACKnowledge
ADC	Analog-to-Digital Converter
ALU	Arithmetic Logic Unit
AS	Application Section
ASIC	Aplication-Specific Integrated Circuit
BLS	Bootloader Section
BS	Back Space
CAN	Controller Area Network
CD-ROM	Compact Disc Read-Only Memory
CMOS	Complementary Metal Oxide Semiconductor
COITT	Colegio Oficial de Ingenieros Técnicos de Telecomunicación
CPU	Central Processing Unit
CR	Carry Return
CRC	Cyclic Redundant Check
DLC	Data Length Code
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
EOF	End Of Frame
FIFO	First In First Out
FPGA	Field Programmable Gate Array

GPIO	General Purpose Input Output
I2C	Inter-Integrated Circuit
IDE	IDentifier Extension
IEEE	Institute of Electrical and Electronics Engineers
IFS	Intermission Frame Space
IGIC	Impuesto General Indirecto Canario
ISO	International Organization for Standardization
JTAG	Joint Test Action Group
LF	Line Feed
LLC	Logical Link Control
MAC	Medium Access Control
MCU	MicroController Unit
MIPS	Million Instructions Per Second
MOB	Message Object
OSI	Open Systems Interconnection
PC	Program Counter
PCB	Printed Circuit Board
PLS	Physical Signalling
PWM	Pulse Width Modulation
QFN	Quad Flat No Leads
RAM	Random Access Memory
RF	Radio Frecuencia
RISC	Reduced Instruction Set Computer

RTC	Real Time Counter
RTR	Remote Transmission Request
SOF	Start Of Frame
SPI	Serial Peripheral Interface
SPM	Store Program Memory
SRAM	Static Random Access Memory
TFG	Trabajo Fin de Grado
TQFP	Thin Quad Flat Pack
ULPGC	Universidad de Las Palmas de Gran Canaria
USB	Universal Serial Bus
USART	Universal Synchronous Asynchronous serial Receiver and Transmitter
WSN	Wireless Sensor Network



# Capítulo 1:

# Introducción

## 1.1 Antecedentes.

Los avances en la tecnología de sistemas micro-electrónicos, comunicaciones inalámbricas, y electrónica digital han posibilitado el desarrollo de nodos sensores de bajo coste, reducido consumo de potencia y mínimo tamaño, que se comunican a distancia. Estos pequeños nodos sensores inalámbricos se fundamentan en la idea de redes de sensores basadas en el esfuerzo conjunto de un gran número de nodos [1, 2].

Una red de sensores inalámbricos (*Wireless Sensor Network, WSN*) consiste en una serie de nodos sensores distribuidos espacialmente con el objetivo de medir magnitudes físicas, como pueden ser sonido, temperatura, presión, etc. Dichos nodos sensores se comunican entre sí de forma inalámbrica con el fin de transmitir la información medida, por lo general a una unidad de procesamiento central, o bien para llevar a cabo algunas tareas de coordinación, como se representa en la Figura 1.1. Los componentes hardware más usuales en un nodo sensor incluyen un procesador empotrado, un transceptor de radio, fuente de alimentación y uno o más sensores [1, 2].

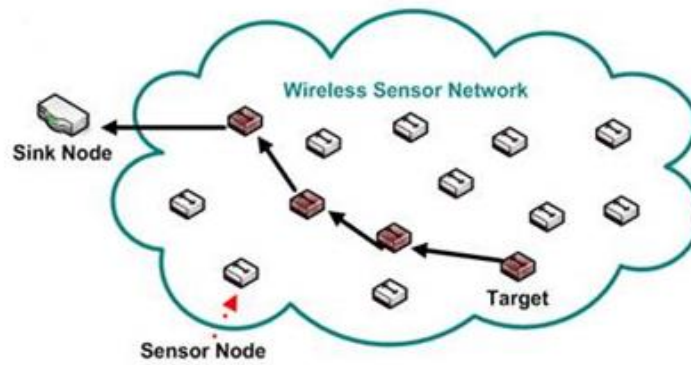


Figura 1.1. Wireless Sensor Network

En un nodo sensor, la funcionalidad del procesador empujado es la de programar tareas, procesar datos y controlar el funcionamiento de los otros componentes hardware. Los tipos de procesadores empujados que pueden ser usados en un nodo sensor incluyen Microcontroladores (*MicroController Unit*, MCU), *Digital Signal Processor* (DSP), *Field Programmable Gate Array* (FPGA) y *Application-Specific Integrated Circuit* (ASIC). De entre todas estas alternativas, los Microcontroladores han sido los más usados en la práctica debido a su flexibilidad para conectar con otros dispositivos, así como su reducido precio [3].

El transceptor es el responsable de la comunicación inalámbrica del nodo sensor. Las opciones disponibles como medio de comunicación inalámbrico son Radio Frecuencia (RF), Láser, e Infrarrojo. De ellas las comunicaciones basadas en RF encajan mejor en la mayoría de aplicaciones de WSN [3].

En un nodo sensor, el consumo de energía se produce principalmente debido a las tareas de detección, comunicación y procesamiento de los datos. Así, se requiere mayor energía para la comunicación que para el resto de las tareas, siendo las baterías la principal fuente de energía para los nodos sensores, por lo que, debido a su limitada capacidad,

minimizar el consumo de energía siempre es un aspecto clave durante las operaciones de los nodos en las WSNs [3].

Un sensor es un dispositivo hardware que ante un cambio en una magnitud física, como pueden ser temperatura, presión o humedad, genera como respuesta una señal eléctrica medible. La señal eléctrica analógica detectada por los sensores es digitalizada por un conversor analógico-digital (*Analog-to-Digital Converter*, ADC) y enviada al procesador empotrado para un procesamiento posterior. Debido a que un nodo sensor es un dispositivo micro-electrónico alimentado por una fuente de energía limitada, los sensores acoplados deben ser igualmente de reducido tamaño y tener muy poco consumo de energía. Un nodo sensor puede tener varios tipos de sensores integrados dentro, o conectados al nodo [3].

En los últimos años, las redes de sensores inalámbricos han ganado mucha popularidad, experimentando un gran desarrollo y despliegue acrecentado conforme al aumento de las capacidades tecnológicas de fabricación de sistemas integrados y al desarrollo de nuevos estándares y estructuras de comunicación. La base de este auge proviene de maximizar la utilización de una de sus principales ventajas: la movilidad. Haciendo uso de este tipo de redes, se elimina la necesidad de usar cables para la transferencia de datos entre dispositivos, dotando de una gran flexibilidad a la red, y por tanto, incrementando su eficiencia y productividad, independientemente del lugar donde ésta se encuentre instalada. La flexibilidad en el despliegue es una propiedad fundamental de los dispositivos que empleen este tipo de redes, y que está cada vez más presente en la sociedad actual [1-3].

Las redes de sensores inalámbricos presentan un gran potencial para aplicaciones en escenarios como el seguimiento y la vigilancia de objetivos militares, atenuación de desastres naturales, monitorización biomédica de la salud, exploración en entornos

## | Introducción

peligrosos y detección de sismos. En el seguimiento y vigilancia de objetivos militares, una WSN puede ayudar en la detección de intrusos e identificación. En desastres naturales, los nodos sensores pueden detectar el entorno para predecir desastres con antelación a que ocurran. En aplicaciones biomédicas, la implantación quirúrgica de sensores ayudan a monitorizar la salud del paciente. Para la detección de sismos, el despliegue de sensores *ad-hoc* a lo largo del área volcánica puede detectar el desarrollo de terremotos y erupciones [2].

Sin embargo, este tipo de redes también presentan una serie de desventajas, siendo una de ellas el que, a pesar de que emplean tecnología inalámbrica, la programación y configuración de los nodos debe realizarse por lo general con el dispositivo físicamente conectado. Esto implica que en redes que sean relativamente extensas puede resultar muy costoso en tiempo el ir programando nodo por nodo, sin tener en cuenta que la red puede estar desplegada en una superficie de gran tamaño, por lo que se requerirá estar desplazándose de un nodo a otro para realizar dicha programación [4, 5].

De modo similar, se hace muy difícil la depuración y el control de la WSN, ya que cuando se detecta que ocurre algún fallo en alguno de los nodos, la única forma que habría de comprobar qué es lo que está pasando, es desplazarse al lugar donde se encuentra dicho nodo y realizar una depuración *in situ*. Las desventajas que esto trae consigo son evidentes, ya que puede representar un trabajo muy engorroso tener que ir de nodo en nodo verificando su funcionamiento. Si además se dispone de una red ya desplegada, el proceso de detectar el error, corregirlo y verificar nuevamente el funcionamiento, lo que seguramente será necesario realizar varias veces, se puede fácilmente estimar que consumirá un tiempo significativo [4, 5].

Actualmente existen alternativas que permiten programar los nodos de forma inalámbrica, si bien esta solución no es demasiado eficaz, ya que si ocurre algún problema



con la transmisión, dicha programación no se llevará a cabo y no será posible saber qué es lo que ha pasado. En este trabajo se pretende dar una solución a los problemas comentados anteriormente con el desarrollo de una red experimental de sensores inalámbricos [4, 5].

## 1.2 Objetivos.

El objetivo principal del presente Trabajo Fin de Grado (TFG) consiste en desarrollar una infraestructura que facilite la experimentación en el desarrollo de redes de sensores inalámbricos. Una de las principales características de la red experimental de sensores inalámbricos relacionada con este trabajo será que estos sensores estarán conectados mediante un bus común a un Servidor central (*Host*), siendo el bus utilizado CAN (*Controller Area Network*) (Figura 1.2).

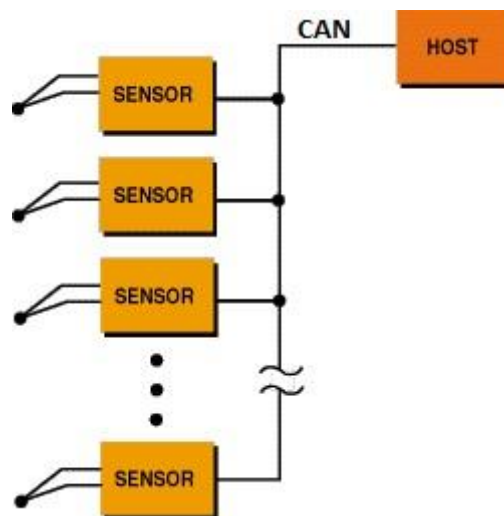


Figura 1.2. Configuración en bus

Los sensores proporcionarán constantemente información de su estado al servidor, con lo cual se busca facilitar las tareas de depuración y control que pueden resultar complejas en caso de que la red no esté cableada, ya que habría que ir nodo por nodo verificando su funcionamiento. Además, dichos informes de los nodos se almacenarán en

## | Introducción

el Servidor, permitiendo así verificar su funcionamiento en el momento en el que se desee, no necesariamente en tiempo real.

El tener la red cableada a través del bus permite programar cada nodo con un *firmware* de forma remota desde el Servidor, pudiendo desarrollar distintas versiones del mismo, y probarlas en diferentes nodos. Esto permite también que más de un usuario utilice la red al mismo tiempo, ya que al tener cada nodo un identificador, estos se pueden programar individualmente.

Por otro lado, al estar todos los datos concentrados en un servidor, es posible conectar dicho servidor a Internet, de tal forma que los usuarios de la red puedan acceder a ella de forma remota a través de un navegador web. Esta característica permite que cualquier usuario pueda acceder a la red sin estar físicamente en la misma ubicación, programe los sensores con el *firmware* que haya desarrollado, y finalmente analice los resultados obtenidos en el momento que desee, todo esto sin tener que desplazarse.

Así, en este TFG se van a implementar las funciones básicas de programación y monitorización que permitan crear una plataforma experimental que facilitará la realización de pruebas en redes de sensores inalámbricos de una manera mucho más práctica de cara al usuario. Facilitará la tarea del programador, ya que éste sólo tendrá que centrarse en el desarrollo del *firmware* y no en la implementación de la red de prueba. Además, ofrecerá varias ventajas frente a las redes completamente inalámbricas a la hora de programar los nodos, realizar la depuración, recoger los resultados, y facilidad de acceso a la red. También será muy escalable, ya que será posible realizar pruebas iniciales en unos pocos nodos y luego pasar a una red masiva de elementos sin ninguna dificultad. Otro objetivo con el que se ha pensado este TFG es de cara a la docencia y las prácticas de los estudiantes de la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), ya que permitirá al profesor asignarle

unos determinados nodos a cada alumno. Estos podrán realizar sus prácticas en dichos nodos y el profesor podrá analizar los resultados obtenidos por cada alumno, dado que estarán almacenados en el Servidor, permitiendo así una evaluación más precisa de las prácticas.

Para el desarrollo de este TFG se parte de unos módulos sensores inalámbricos que han sido desarrollados previamente por los profesores tutores en su labor de investigación, siendo estos los nodos sobre los cuales se trabajará. Una característica de estos módulos es que están basados en el microcontrolador AT90CAN128 de Atmel, el cual posee una interfaz CAN, además de una sección de memoria y funcionalidades destinadas especialmente a la implementación del *bootloader*, características convenientes para llevar a cabo el presente TFG.

### 1.3 Solución adoptada.

En la Figura 1.3 se puede ver un diagrama de bloques de la solución adoptada. En él se representan los diferentes elementos que componen el sistema y en los cuales se basa este Trabajo Fin de Grado.

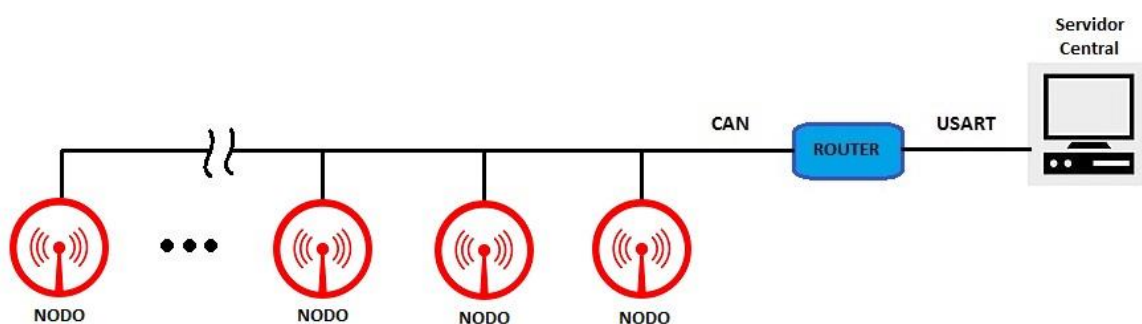


Figura 1.3. Estructura de la plataforma

## | Introducción

En primer lugar, en el Servidor central se toma el código fuente desarrollado, se analiza y se divide en pequeños fragmentos de código. En segundo lugar se selecciona la dirección del nodo que se desea programar y se envía al *router* un comando que fija esa dirección. El tercer paso consiste en tomar los fragmentos de código, formar con ellos los distintos comandos que posteriormente deben interpretar los nodos, y enviarlos al *router*. En cuarto lugar el *router* recibe los comandos del servidor a través de la USART (*Universal Synchronous Asynchronous serial Receiver and Transmitter*), los convierte en tramas CAN, y los envía por este bus a la dirección previamente establecida. En quinto lugar los nodos, una vez hayan recibido un comando, lo interpretan y llevan a cabo la tarea correspondiente. Por último, cualquiera de los nodos puede enviar mensajes al Servidor a través del *router*; si este es el caso el *router* toma la trama CAN recibida, la convierte en un mensaje serie, y lo envía al servidor a través de la USART.

## 1.4 Estructura de la memoria.

La memoria de este Trabajo Fin de Grado está estructurada en cinco capítulos. Aparte se encuentra incluida una lista de Acrónimos, situada al comienzo del documento, así como una lista de Referencias Bibliográficas, Pliego de Condiciones, el Presupuesto y los Anexos que se encuentran localizados al final de la memoria.

El primer capítulo presenta una introducción a las redes de sensores inalámbricos, incluyendo una descripción general de los elementos que la componen, así como de sus principales ventajas, campos de aplicación y problemas relativos al desarrollo de *firmware* para este tipo de redes. A continuación se plantean los objetivos de este Trabajo Fin de Grado, describiendo brevemente la plataforma experimental en la que se engloba el desarrollo a realizar.

En el segundo capítulo se realiza una descripción del hardware utilizado, haciendo especial hincapié en el microcontrolador Atmel AT90CAN128, elemento principal en la realización de este Trabajo Fin de Grado. De él se describen sus características más importantes, prestando especial atención a aquellas que se han utilizado durante el desarrollo del proyecto, y que básicamente son el controlador CAN, la USART, la zona de *Bootloader*, y los pines GPIO (*General Purpose Input Output*).

En el tercer capítulo se comienza con una breve descripción general de la plataforma. A continuación se explica detalladamente el sistema de interrupciones empleado. Además también se introduce la biblioteca CAN desarrollada como parte de este Trabajo Fin de Grado, que será puesta a disposición de los usuarios. Por último, se describe el funcionamiento de la aplicación *router* y la aplicación *bootloader* implementadas, explicando detalladamente cada una de las partes en las que se divide el programa ejecutado por el microcontrolador.

En el cuarto capítulo, tras haber completado el desarrollo *firmware*, se describen las pruebas realizadas para comprobar el correcto funcionamiento de la plataforma.

El quinto capítulo está dedicado a las conclusiones obtenidas tras haber completado los objetivos propuestos para el presente Trabajo Fin de Grado, así como a las líneas futuras.



# Capítulo 2:

## Descripción Hardware

### 2.1 Introducción.

En este capítulo se pretenden introducir los distintos elementos hardware que componen la plataforma experimental sobre la que se basa el presente TFG. Se comienza con una descripción general de los módulos sensores inalámbricos, haciéndose hincapié en las partes relativas a este TFG. A continuación se presenta una introducción del microcontrolador Atmel AT90CAN128, el cual ha sido empleado en este Trabajo Fin de Grado, realizándose en primer lugar una descripción general del mismo para finalmente, centrarse en los bloques de los cuales se ha hecho uso.

### 2.2 Módulos sensores inalámbricos.

Para el desarrollo de este TFG se han utilizado unos módulos sensores inalámbricos que han sido desarrollados previamente por los profesores tutores en su labor de investigación, siendo los dispositivos sobre los cuales se trabajará. Una característica de estos módulos es que están basados en el microcontrolador AT90CAN128 de Atmel. En la Figura 2.1 se puede observar la placa de circuito impreso (*Printed Circuit Board, PCB*) de estos módulos.

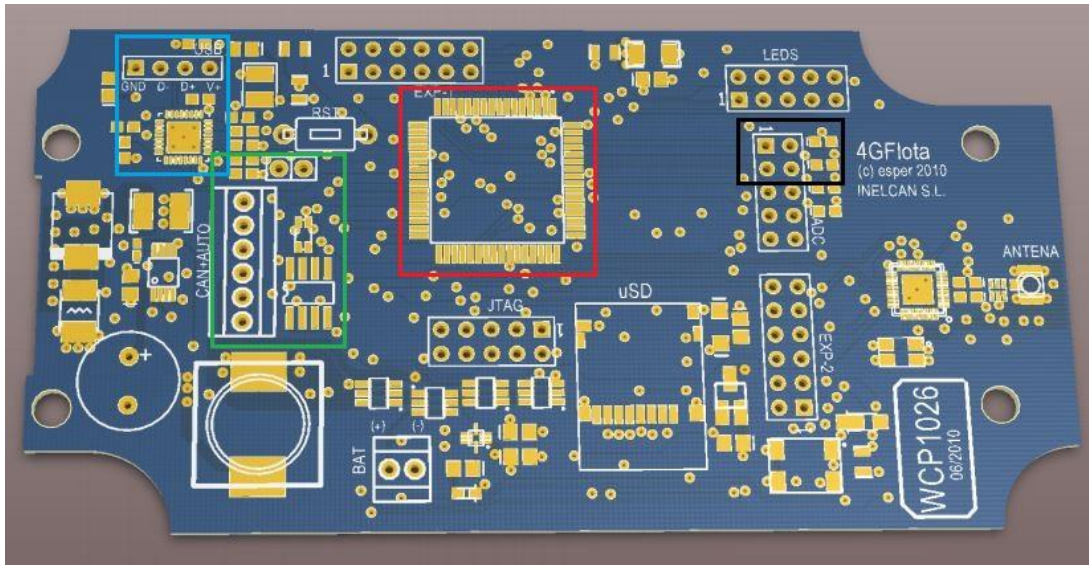


Figura 2.1. PCB de los módulos sensores inalámbricos

Enmarcados con los distintos bloques de colores se han señalado los elementos que intervienen directamente en este TFG. En color rojo se ha señalado la zona donde va colocado el microcontrolador AT90CAN128. El color negro señala el pin con el cual se indica al microcontrolador que se debe iniciar en modo *bootloader*. El color azul corresponde a la conexión USB (*Universal Serial Bus*) con su correspondiente interfaz hacia la USART. Por último en color verde están los elementos relacionados con el CAN.

El pin para seleccionar el modo *bootloader* se encuentra ubicado en uno de los puertos de expansión de la placa. Colocando un *jumper* entre este pin y masa, se le indica al microcontrolador que se debe iniciar en modo *bootloader*, en caso contrario saltará directamente a la aplicación de usuario.

El conector USB está compuesto por cuatro pines. Uno correspondiente a la alimentación, otro para masa y los dos restantes corresponden al par diferencial de datos. A través de este conector se puede alimentar a toda la placa. La señal de datos se lleva a un chip que hace de interfaz entre USB y la USART. El FT232R es un chip que simplifica los



diseños USB a serie, reduce el número de componentes externos necesarios al integrar en el dispositivo una EEPROM, resistencias de terminación USB y un circuito integrado de reloj que no requiere de cristal externo. En la Figura 2.2 se presenta una imagen del chip FT232R en su formato QFN (Quad Flat No Leads) [6].



Figura 2.2. Chip FT232R

En la Figura 2.3 se puede observar el esquemático de cómo está conectado este chip con el microcontrolador y con el conector USB. Por un lado se tiene la señal de datos USB (USBDP, USBDM), que una vez pasa por la interfaz se convierte en las señales del transmisor y receptor de la USART (TXD, RXD).

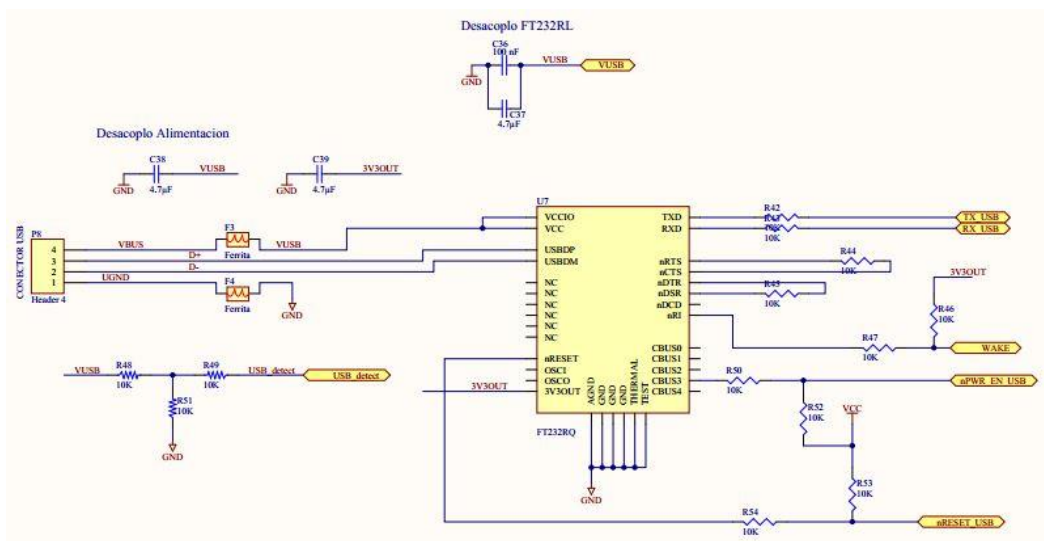


Figura 2.3. Esquemático del circuito USB a UART

## Descripción Hardware

El microcontrolador AT90CAN128 dispone de un controlador pero no de un transceptor CAN, por lo tanto es necesario añadirlo de forma externa. El dispositivo utilizado en este caso es el chip SN65HVD255, del cual se puede observar una imagen en la Figura 2.4. Este es un chip que cumple con los requerimientos del estándar ISO 11898, pudiendo soportar hasta 1Mbps de velocidad en buses altamente cargados y proporciona varias características de protección que permiten aumentar la robustez del bus [7].



Figura 2.4. Chip SN65HVD255

En la Figura 2.5 se puede observar la posición del transceptor externo que se ha añadido en los módulos, así como la de los pines correspondientes a las conexiones de CAN\_H y CAN\_L, en los cuales se ha colocado una clema para facilitar su conexionado.

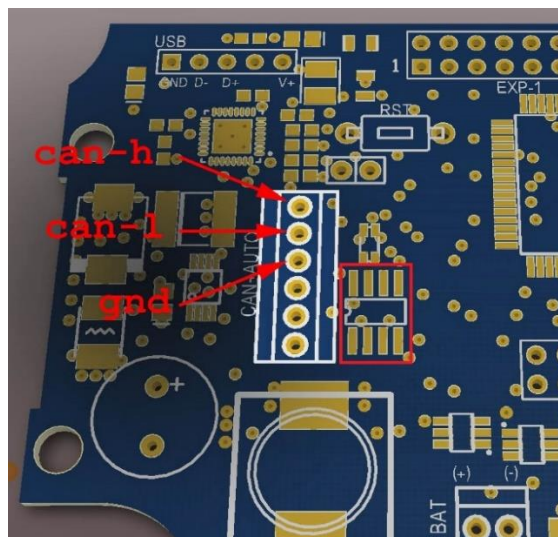


Figura 2.5. Conector CAN

Por último, se incluye en la Figura 2.6 el esquemático de cómo se ha realizado el conexionado del transceptor. En ella se puede observar cómo por un lado se tienen las señales de transmisión y recepción CAN del microcontrolador que una vez pasan por el transceptor se convierten en CAN\_H y CAN\_L como establece el estándar. Destacar también que se dispone de un *jumper* con el cual se establece si se añade o se quita la terminación resistiva, dependiendo si se trata de un nodo final o uno intermedio respectivamente.

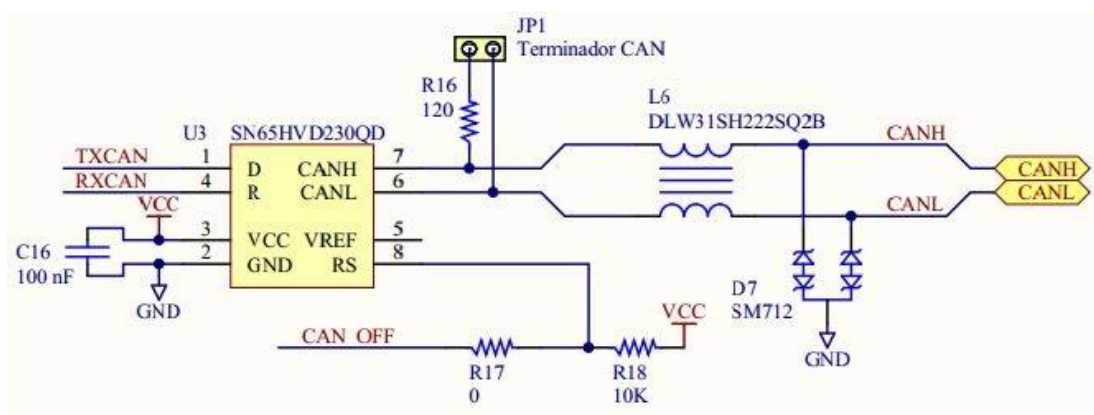


Figura 2.6. Esquemático del circuito transceptor del CAN

## 2.3 Microcontrolador Atmel AT90CAN.

### 2.3.1 Descripción general.

El microcontrolador Atmel AT90CAN32/64/128 es un microcontrolador CMOS de 8 bits de propósito general y bajo consumo [8]. El diseño de su núcleo está basado en la arquitectura RISC (*Reduced Instruction Set Computer*) de AVR. La arquitectura del núcleo AVR combina la utilización de un amplio repertorio de instrucciones con un banco de 32 registros de propósito general. Estos 32 registros están directamente conectados a la ALU (*Arithmetic Logic Unit*), permitiendo realizar operaciones entre dos registros independientes en un sólo ciclo de reloj.

## | Descripción Hardware

En este TFG se hará uso de la versión AT90CAN128, la cual consta de 128KBytes de memoria de programa, implementados en una memoria tipo *Flash*, 4KBytes de memoria tipo EEPROM (*Electrically Erasable Programmable Read-Only Memory*), 4KBytes de memoria SRAM (*Static Random Access Memory*). Además cuenta con 53 líneas de entrada/salida de propósito general (*General Purpose Input Output, GPIO*), 32 registros de propósito general, un controlador CAN, contador de tiempo real (*Real Time Counter, RTC*), 4 Timers/Contadores flexibles con modo de comparación y PWM (*Pulse Width Modulation*). También cuenta con hardware USART (*Universal Synchronous Asynchronous serial Receiver and Transmitter*), hardware SPI (*Serial Peripheral Interface*), una interfaz serie de dos cables (*Inter-Integrated Circuit, I2C*), conversor analógico-digital (*Analog to Digital Converter, ADC*) de 8 canales y resolución de 10 bits con la opción de incluir una etapa de entrada diferencial con ganancia programable, *Watchdog* programable con oscilador interno, interfaz JTAG (*Joint Test Action Group*) para test conforme al estándar IEEE 1149.1, también usada para acceder al sistema de depuración en chip y la programación.

El sistema está dotado con 5 modos distintos de ahorro de energía. El modo *Idle* detiene la CPU (*Central Processing Unit*) mientras permite que la RAM, *timer/counters*, puertos SPI/CAN y el sistema de interrupciones siga funcionando. El modo *Power-down* almacena el contenido de los registros pero detiene el oscilador, deshabilitando el resto de funciones del chip hasta la próxima interrupción o *reset*. En el modo *Power-save*, el *timer* asíncrono continúa contando, permitiendo al usuario mantener un *timer* de referencia, mientras que el resto del dispositivo está inactivo. El bloque conversor analógico-digital dispone de un modo especial de operación, *ADC Noise Reduction*, que desconecta todo el hardware del sistema, excepto el contador asíncrono, con el fin de realizar una conversión fiable, aportando el mínimo ruido posible a la medida realizada. El último modo es el *Standby*, en el que el oscilador, ya sea basado en cristal o un resonador, continúa operando mientras que el resto del dispositivo está inactivo, lo que permite un arranque muy rápido del dispositivo a la vez que un bajo consumo de potencia.

En la Figura 2.7 se puede observar una imagen del microcontrolador Atmel AT90CAN128 en el formato TQFP (*Thin Quad Flat Pack*), que es el que ha sido utilizado, el cual consta de 64 pines.



Figura 2.7. Atmel AT90CAN128 Microcontroller

### 2.3.2 La arquitectura AVR.

La arquitectura AVR es una modificación de la arquitectura RISC de 8 bits existente para microcontroladores. Fue desarrollada por Atmel en 1996; concretamente, fue inventada por dos estudiantes del *Norwegian Institute of Technology*, y posteriormente refinada y desarrollada en Atmel Norway, la empresa subsidiaria de Atmel, fundada entonces por los dos diseñadores del chip. Los de AVR fueron los primeros microcontroladores que empleaban una memoria *Flash* integrada para almacenar el programa [9-11].

AVR es una CPU de arquitectura Harvard, en la que la memoria de programa y la memoria de datos se encuentran separadas físicamente. Sin embargo, existen una serie de instrucciones especiales que permite el acceso a la memoria de programa durante la ejecución del mismo.

## | Descripción Hardware

La arquitectura AVR está diseñada para soportar frecuencias de reloj de hasta 20MHz, pudiendo llegar algunos dispositivos hasta los 32MHz. Sin embargo, las aplicaciones que requieran un bajo consumo requerirán menores frecuencias de reloj. Algunas series de microcontroladores AVR integran un oscilador en el chip, eliminando así la necesidad de utilizar una fuente de reloj externa, o un circuito resonador. Con el fin de incrementar las posibilidades del sistema, también se incluye un *prescaler* para la señal de reloj, de forma que es posible dividir la frecuencia de reloj principal por un valor máximo de 256. El valor del *prescaler* puede ser configurado en el momento de la programación o durante la ejecución de la aplicación, aportando máxima flexibilidad. Casi todas las instrucciones ejecutadas en el AVR duran 1 ciclo de reloj, lo que permite realizar 1 MIPS (*Million Instructions Per Second*) por MHz.

En cuanto a memoria, tanto *Flash* como EEPROM y SRAM, se encuentran integradas en el hardware disponible en el chip. De este modo se elimina la necesidad de emplear memorias externas. Sin embargo, algunos dispositivos disponen de un bus, que permite hacer uso de una memoria externa, estando ésta mapeada en la memoria del chip.

En la mayoría de las variantes de la arquitectura AVR, los 32 registros de uso general se encuentran mapeados en las primeras 32 direcciones de memoria, y los 64 registros de entrada-salida se encuentran mapeados consecutivamente a los de uso general. De este modo, la concatenación de los 32 registros de uso general con los registros de entrada-salida y la memoria de datos conforman un espacio de direcciones unificado, accesible mediante operaciones de carga/almacenamiento.

Las CPU AVR poseen un *pipeline* simple de 2 etapas (cargar y ejecutar), haciendo posible que la mayor parte de las instrucciones se ejecuten en un solo ciclo de reloj. Esta característica particular hace que esta arquitectura resulte significativamente más rápida frente a otros microcontroladores de 8 bits con otras arquitecturas.

AVR fue diseñado desde un comienzo para la ejecución eficiente de código C compilado. Como este lenguaje utiliza punteros para el manejo de variables en memoria, los tres últimos pares de registros internos del procesador AVR se usan como punteros de 16 bits al espacio de memoria externa, bajo los nombres X, Y y Z. Este compromiso se aplica en arquitecturas de ocho bits debido a que el tamaño de palabra de 8 bits tan solo puede direccionar 256 registros.

### 2.3.3 Diagrama de bloques del microcontrolador AT90CAN128.

En la Figura 2.8 se muestra el diagrama de bloques completo del microcontrolador Atmel AT90CAN128. En él se pueden diferenciar los distintos bloques funcionales, englobados mediante cuadros coloreados atendiendo a la función que desempeñan.

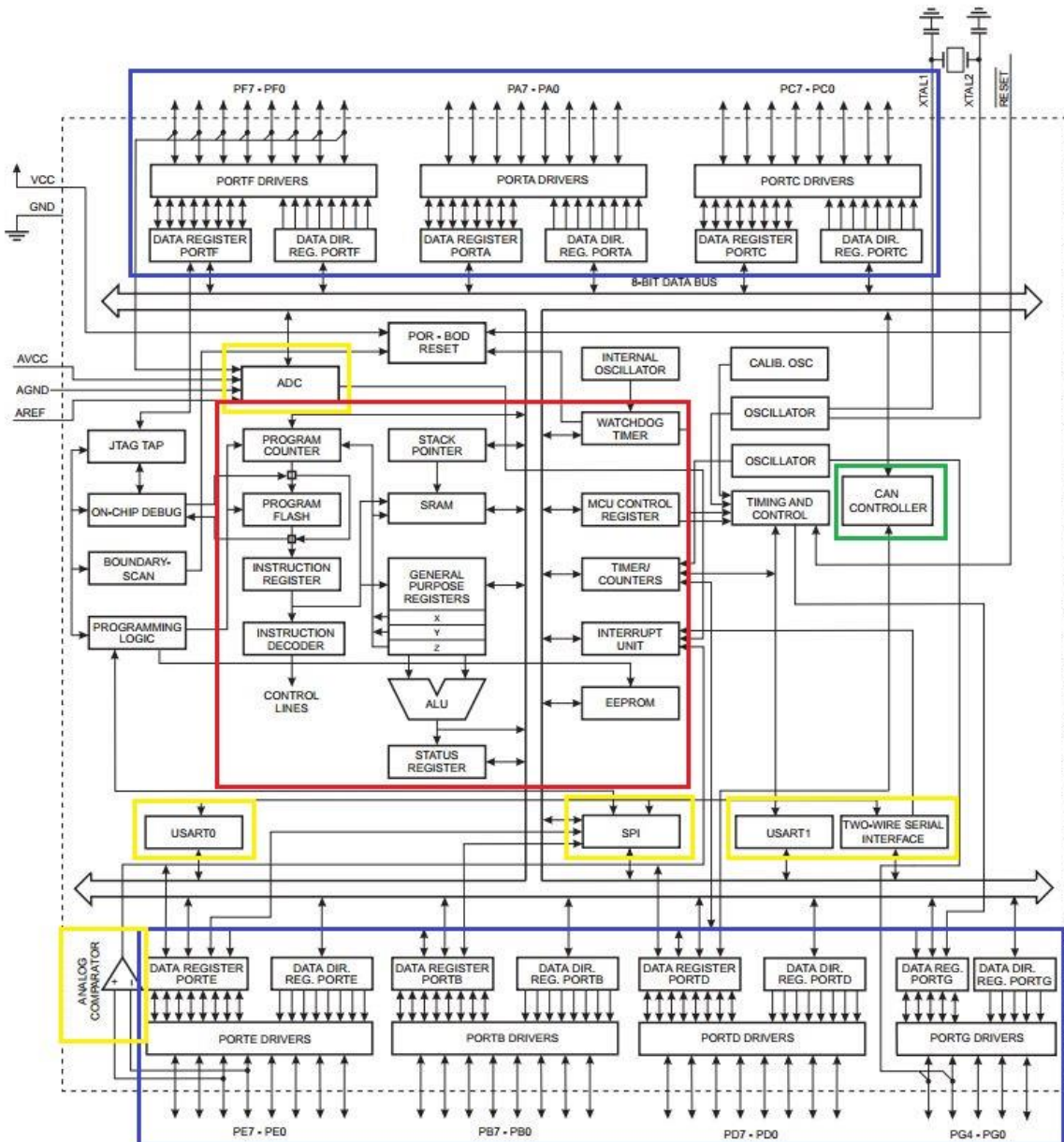


Figura 2.8. Diagrama de Bloques del microcontrolador AT90CAN128

Los bloques encuadrados en azul se encuentran formados por los puertos de entrada y salida del dispositivo. El Atmel AT90CAN128 posee 53 líneas de entrada/salida que pueden ser utilizadas para propósito general, y que se encuentran organizadas en 6 puertos de 8 bits y un puerto de 5 bits, denotados como PORTA, PORTB, PORTC, PORTD, PORTE, PORTF y PORTG, respectivamente. Las líneas de uso general comparten conexión



con los diferentes bloques funcionales implementados en el hardware del dispositivo, de forma que éstos poseen conexión con el exterior.

Los bloques resaltados en color amarillo son algunos de los distintos bloques funcionales implementados en el chip. Dichos bloques poseen una variada funcionalidad, que va desde la USART, SPI, bloque de conversor analógico-digital (ADC), *timers*, etc. Por otro lado el bloque resaltado en color verde se corresponde con el controlador CAN, siendo este el bloque más importante para nuestro TFG.

En la Figura 2.9 se muestra el diagrama de bloques del núcleo del microcontrolador, que se corresponde con el bloque resaltado en rojo en la Figura 2.8.

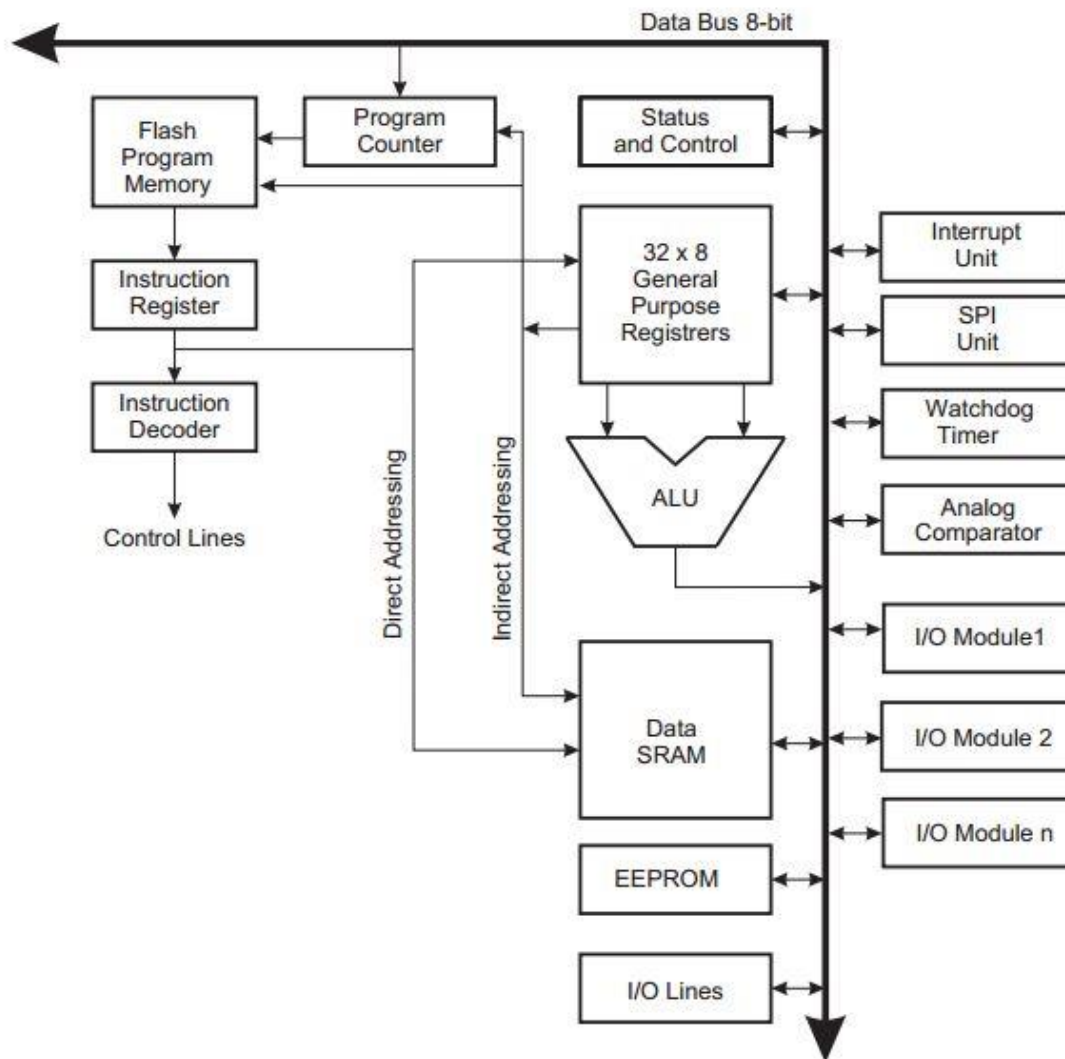


Figura 2.9. Arquitectura AVR

## 2.4 Funcionalidades del microcontrolador AT90CAN128.

En este apartado se describen las características principales de los bloques funcionales más significativos del microcontrolador Atmel AT90CAN128, y que han sido empleados en el desarrollo de este Trabajo Fin de Grado.

### 2.4.1 Puertos de entrada/salida de propósito general (GPIO).

Como se ha mencionado anteriormente, el microcontrolador AT90CAN128 cuenta con 53 líneas de entrada/salida de propósito general (GPIO). Dichas líneas se encuentran agrupadas en 6 puertos de 8 bits y un puerto de 5 bits, denotados como PORTA, PORTB, PORTC, PORTD, PORTE, PORTF y PORTG, respectivamente. Además cada una de ellas cuenta con una resistencia de *pull-up* que brinda mayor capacidad de control sobre las líneas GPIO [8].

Todos los puertos tienen la funcionalidad *Read-Modify-Write* cuando se utilizan como puertos de entrada/salida de propósito general. Esto quiere decir que la dirección de uno de los pines del puerto puede ser cambiada sin modificar accidentalmente la del resto de los pines del puerto. Lo mismo ocurre al habilitar o deshabilitar las resistencias de *pull-up*. En la Figura 2.10 se puede observar el esquema simplificado de un pin.

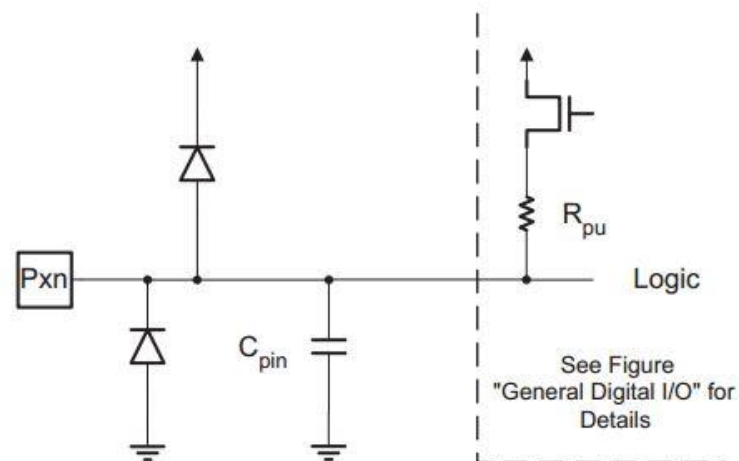


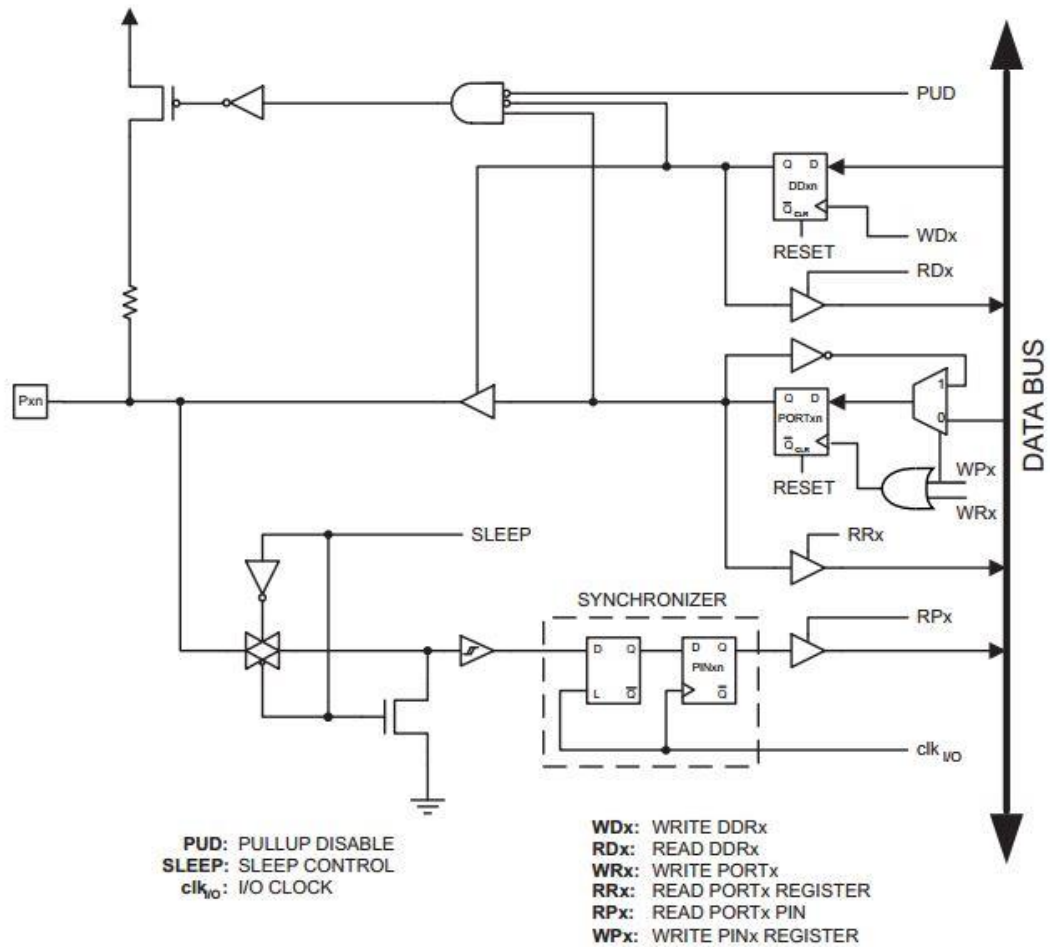
Figura 2.10. Esquemático equivalente de un Pin I/O

Cada puerto tiene reservadas tres posiciones de la memoria de entrada/salida. Una para *Data Register* (PORTx), otra para *Data Direction Register* (DDRx), y otra para *Port Input Pins* (PINx). La letra minúscula “x” representa la letra correspondiente al puerto (A, B, C, D,

E, F y G). El registro *Port Input Pins* es de sólo lectura, mientras que *Data Register* y *Data Direction Register* son de lectura/escritura. Sin embargo, escribir un “1” lógico en uno de los bits del registro PINx resultará en una conmutación en el bit correspondiente en el *Data Register*.

La mayoría de los pines están multiplexados con funciones alternativas de los periféricos implementados en el dispositivo. Aclarar que habilitar en uno de los pines de un puerto la función alternativa, no interfiere con el uso del resto de los pines como entradas-salidas de propósito general.

A continuación, en la Figura 2.11 se presenta el esquema general de un pin cuando éste está actuando como entrada/salida. En él se puede observar que tanto la entrada como la salida se pasan a través de un *latch*, y además que la dirección del pin, ya sea entrada o salida, se controla mediante *buffers* tri-estado.



Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. clk<sub>10</sub>, SLEEP, and PUD are common to all ports.

Figura 2.11. I/O Digital General

### 2.4.1.1 Configuración del Pin.

Cada pin de los puertos consiste en un bit de tres registros distintos: DDxn, PORTxn, y PINxn, siendo la letra minúscula “n” el número correspondiente del pin. Como se muestra en el ejemplo de la Figura 2.12, los bits DDxn se acceden en la dirección DDRx del espacio de direcciones de entrada/salida, los bits PORTxn en la dirección PORTx, y los bits PINxn en la dirección PINx.

**Port G Data Register – PORTG**

Bit	7	6	5	4	3	2	1	0	
	-	-	-	PORTG4	PORTG3	PORTG2	PORTG1	PORTG0	PORTG
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

**Port G Data Direction Register – DDRG**

Bit	7	6	5	4	3	2	1	0	
	-	-	-	DDG4	DDG3	DDG2	DDG1	DDG0	DDRG
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

**Port G Input Pins Address – PING**

Bit	7	6	5	4	3	2	1	0	
	-	-	-	PING4	PING3	PING2	PING1	PING0	PING
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	N/A	N/A	N/A	N/A	N/A	

Figura 2.12. Descripción de registros de los Puertos I/O

El bit DDxn en el registro DDRx selecciona la dirección del pin. Si en DDxn se escribe un “1” lógico, Pxn es configurado como un pin de salida. Mientras que si se escribe un “0” lógico, Pxn se configura como un pin de entrada.

Si en el bit PORTxn del registro PORTx se escribe un “1” lógico cuando el pin se encuentra configurado como entrada, se activa la resistencia de *pull-up*. Para desactivar dicha resistencia, se debe escribir un “0” lógico en PORTxn o el pin debe ser configurado como salida.

Si en el bit PORTxn se escribe un “1” lógico cuando el pin se encuentra configurado como salida, dicho pin se lleva a nivel alto, mientras que si se escribe un “0” lógico el pin se lleva a nivel bajo.

Escribir un “1” lógico en el bit PIN<sub>xn</sub> conmuta el valor de PORT<sub>xn</sub>, independientemente del valor de DDR<sub>xn</sub>, mientras que la lectura de este registro indica el estado del pin.

## 2.4.2 USART.

El microcontrolador AT90CAN128 dispone de dos USART (*Universal Synchronous Asynchronous serial Receiver and Transmitter*), que es un dispositivo de comunicación serie altamente flexible [8, 11]. Los diferentes registros que se explican a continuación se han descrito de forma general, siendo la letra minúscula “n” el número de la USART correspondiente, en este caso 0 o 1. Sus principales características son:

- Modo de operación *Full-Duplex*. (Registros independientes para la transmisión y recepción)
- Funcionamiento síncrono o asíncrono.
- Generador de la tasa de transferencia de alta resolución.
- Soporta tramas serie con 5, 6, 7, 8 y 9 bits de datos, y 1 o 2 bits de *stop*.
- Generación de la paridad y chequeo de paridad soportado por hardware.
- Filtrado de ruido que incluye detección de falsos bits de *start* y filtro digital paso bajo.
- Tres vectores de interrupción independientes para transmisión completada, buffer de transmisión vacío y recepción completada.

En la Figura 2.13 se presenta un diagrama de bloques simplificado de la USART. Los registros y pines accesibles por la CPU se muestran en negrita.

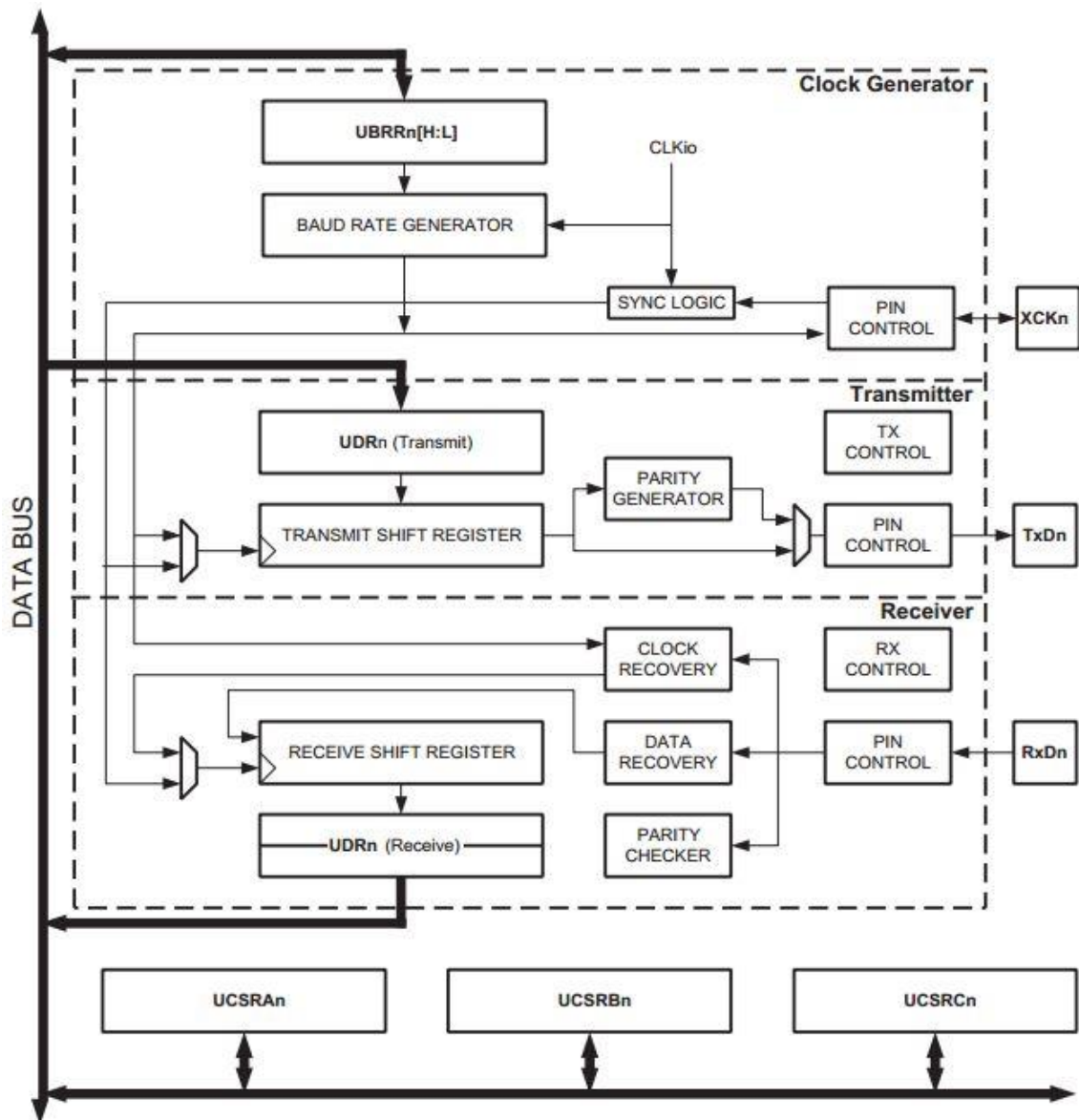


Figura 2.13. Diagrama de bloques de la USART

Los cuadros señalados con líneas discontinuas separan los tres bloques principales de la USART, mencionándolos desde arriba hacia abajo: Generador de Reloj, Transmisor y Receptor. Los registros de control son compartidos por todas las unidades. La lógica del Generador de Reloj (*Clock Generator*) consiste en lógica de sincronización para la entrada de señal de reloj externa utilizada como esclavo en modo síncrono, y el generador de la tasa de baudios. El pin  $XCKn$  (Reloj de transferencia) se usa sólo en el modo de transferencia



síncrono. El Transmisor (*Transmitter*) consiste en un único buffer de escritura (UDRn), un registro de desplazamiento serie (*Shift Register*), un generador de paridad (*Parity Generator*) y lógica de control (*TX Control*) para gestionar diferentes formatos de trama. El buffer de escritura permite una transferencia continua de datos sin retardo entre tramas. El Receptor (*Receiver*) es el bloque más complejo de la USART debido a sus unidades de recuperación de reloj (*Clock Recovery*) y de datos (*Data Recovery*). Estas unidades se utilizan para la recepción asíncrona de datos. El Receptor además incluye un comprobador de paridad (*Parity Checker*), lógica de control (*RX Control*), un registro de desplazamiento (*Shift Register*) y un buffer de recepción (UDRn). El Receptor soporta el mismo formato de tramas que el Transmisor, además puede detectar errores de trama (*Frame Error*), desbordamiento de datos (*Data OverRun*) y errores de paridad (*Parity Errors*).

#### 2.4.2.1 Generación de Reloj.

La lógica de generación de la señal de reloj genera el reloj de referencia para el Transmisor y el Receptor. La USART soporta cuatro modos de operación: Normal asíncrono, Asíncrono a doble velocidad, Maestro síncrono y Esclavo síncrono. El bit UMSELn en el registro de estado y control C de la USART (*Control and Status Register, UCSRnC*) selecciona entre los modos de operación síncrono y asíncrono. En la Figura 2.14 se muestra el diagrama de bloques del generador de reloj de la USART.

## Descripción Hardware

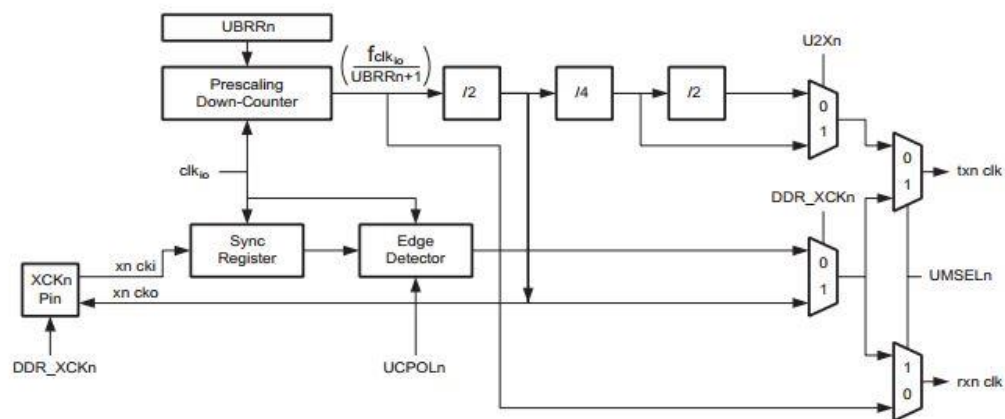


Figura 2.14. Lógica de generación de la señal de reloj de la USART

Cuando se utiliza el modo síncrono ( $UMSELn = 1$ ), el registro DDR para el pin XCKn ( $DDR\_XCKn$ ) controla si la señal de reloj es interna (Maestro) o externa (Esclavo). El pin XCKn sólo está activo cuando se utiliza el modo de operación síncrono. La generación interna del reloj se utiliza para los modos de operación asíncrono y maestro síncrono.

El registro de tasa de baudios (*Baud Rate Register*,  $UBRRn$ ) y el contador regresivo conectado a él funcionan como un *prescaler* programable. El contador regresivo, ejecutándose a la velocidad del reloj del sistema ( $f_{clkio}$ ), se carga con el valor de  $UBRRn$  cada vez que el contador llega a cero, o bien cuando se escribe en el registro  $UBRRn$ . Se genera un pulso de reloj cada vez que el contador llega al valor cero. Este reloj es la salida del generador de la tasa de baudios ( $= f_{clkio} / (UBRRn + 1)$ ). El Transmisor divide la salida de este reloj por 2, 8 o 16 dependiendo del modo de operación. Por otro lado este reloj es usado directamente por las unidades de recuperación de datos y reloj del Receptor. Sin embargo, las unidades de recuperación utilizan una máquina de estados que usa 2, 8 o 16 estados en función del modo de operación establecido. En la Figura 2.15 se muestran las distintas ecuaciones para calcular tanto la tasa de baudios como el valor del registro  $UBRRn$ .

Operating Mode	Equation for Calculating Baud Rate <sup>(1)</sup>	Equation for Calculating UBRRn Value
Asynchronous Normal mode (U2Xn = 0)	$BAUD = \frac{f_{CLKio}}{16(UBRRn + 1)}$	$UBRRn = \frac{f_{CLKio}}{16BAUD} - 1$
Asynchronous Double Speed mode (U2Xn = 1)	$BAUD = \frac{f_{CLKio}}{8(UBRRn + 1)}$	$UBRRn = \frac{f_{CLKio}}{8BAUD} - 1$
Synchronous Master mode	$BAUD = \frac{f_{CLKio}}{2(UBRRn + 1)}$	$UBRRn = \frac{f_{CLKio}}{2BAUD} - 1$

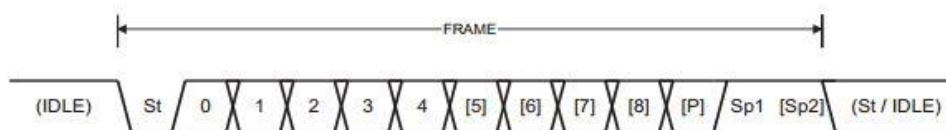
Note: 1. The baud rate is defined to be the transfer rate in bit per second (bps)

- BAUD** Baud rate (in bits per second, bps).
- f<sub>clk<sub>io</sub></sub>** System I/O Clock frequency.
- UBRRn** Contents of the UBRRnH and UBRRnL Registers, (0-4095).

Figura 2.15. Ecuaciones para calcular la configuración de la tasa de baudios

### 2.4.2.2 Formato de la trama.

Una trama serie se define como una serie de bits de datos en conjunto con bits de sincronización (bits de *start* y *stop*). Opcionalmente se puede incluir un bit de paridad para comprobación de errores. En la Figura 2.16 se observa gráficamente este formato, donde los bits dentro de corchetes son opcionales.



- St** Start bit, always low.
- (n)** Data bits (0 to 8).
- P** Parity bit. Can be odd or even.
- Sp** Stop bit, always high.
- IDLE** No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.

Figura 2.16. Formato de trama

## Descripción Hardware

La USART acepta las 30 combinaciones de los siguientes formatos de trama válidos:

- 1 bit de *start*.
- 5, 6, 7, 8 o 9 bits de datos.
- Bit de paridad par, impar o sin paridad.
- 1 o 2 bits de stop.

Una trama comienza con el bit de *start* seguido del bit de datos menos significativo. A continuación se encuentran los siguientes bits de datos, hasta un máximo de 9, terminando con el bit más significativo. Si está habilitado, el bit de paridad se inserta justo después de los bits de datos, antes de los bits de *stop*. Cuando una trama es transmitida completamente, se puede transmitir directamente una nueva trama, o la línea de comunicación se puede llevar a un estado desocupado (nivel alto).

### 2.4.2.3 Transmisor.

La USART debe ser inicializada antes de que pueda tener lugar cualquier comunicación. El proceso de inicialización normalmente consiste en el establecimiento de la tasa de baudios, el formato de la trama, y en la habilitación del Transmisor o del Receptor dependiendo del uso que se le vaya a dar. Si la USART va a ser gestionada mediante interrupciones, el *flag* de interrupciones globales (*Global Interrupt Flag*) debe estar desactivado, y las interrupciones globales deshabilitadas, en el momento de hacer la inicialización.

El Transmisor de la USART se habilita al establecer el bit de habilitar transmisión (*Transmit Enable*, TXENn) en el registro UCSRnB a nivel alto. Una vez habilitado, el modo de operación normal del puerto en el pin TxDn es ignorado por la USART y se le asigna la función de salida serie del Transmisor. La tasa de baudios, el modo de operación y el formato de la trama deben ser establecidos antes de hacer ninguna transmisión.

Una transmisión se inicia cargando los datos a transmitir en el buffer de transmisión. Los datos que se encuentran en el buffer serán movidos al Registro de Desplazamiento cuando éste se encuentre listo para enviar una nueva trama. El Registro de Desplazamiento se carga con los nuevos datos si se encuentra en un estado inactivo (no hay transmisiones en curso), o inmediatamente después de que se haya transmitido el último bit de *stop* de la trama anterior.

El Transmisor de la USART cuenta con dos *flags* que indican su estado: registro de datos vacío (*USART Data Register Empty, UDREn*) y transmisión completada (*Transmit Complete, TXCn*). Ambos *flags* pueden ser utilizados para generar interrupciones.

El *flag* de registro de datos vacío (*UDREn*) indica si el buffer de transmisión está listo para recibir nuevos datos. Este bit se activa cuando el buffer está vacío, y se desactiva cuando contiene datos para transmitir que no han sido pasados aún al Registro de Desplazamiento.

El *flag* de transmisión completada (*TXCn*) se activa cuando en el Registro de Desplazamiento la trama completa ha sido desplazada hacia la salida y no hay datos nuevos en el buffer de transmisión. El *flag* *TXCn* es automáticamente desactivado cuando la interrupción de transmisión completada se ejecuta, si bien también puede ser desactivado escribiendo un "1" lógico en el bit correspondiente al *flag*. Cuando el bit de habilitación de la interrupción por transmisión completada (*Transmit Complete Interrupt Enable, TXCIEn*) en el registro *UCSRnB* se pone a nivel lógico alto, se ejecutará dicha interrupción siempre que el *flag* *TXCn* se active, asumiendo que las interrupciones globales está habilitadas.

La acción de deshabilitar el Transmisor (establecer el bit *TXENn* a nivel bajo) no será efectiva hasta que la transmisión en curso y las pendientes se completen, por ejemplo

cuando el Registro de Desplazamiento y el buffer de transmisión no contengan datos para transmitir. Una vez deshabilitado, el Transmisor no continuará sobrescribiendo la función del pin TxDn.

#### 2.4.2.4 Receptor.

El Receptor de la USART se habilita al establecer el bit de habilitar recepción (*Receive Enable*, RXENn) en el registro UCSRnB a "1" lógico. Una vez habilitado, el modo de operación normal del puerto en el pin RxDn es ignorado por la USART y se le asigna la función de entrada serie del Receptor. La tasa de baudios, el modo de operación y el formato de la trama deben ser establecidos antes de realizar ninguna transmisión.

El Receptor comienza la recepción de datos cuando detecta un bit de *start* válido en el bus. Cada bit que le sigue al bit de *start* será muestreado a la velocidad de la tasa de baudios, y luego será desplazado hacia el Registro de Desplazamiento de recepción hasta que se reciba el primer bit de *stop* de la trama. Un segundo bit de *stop* será ignorado por el receptor. Cuando se recibe el primer bit de *stop*, el Registro de Desplazamiento contiene una trama completa, la cual será trasladada al buffer de recepción. El buffer de recepción puede ser leído en la dirección UDRn.

El Receptor de la USART cuenta con un *flag* que indica su estado: recepción completada (*Receive Complete*, RXCn), el cual puede generar una interrupción.

El *flag* recepción completada (RXCn) es un indicador de si hay datos sin leer en el buffer de recepción. El *flag* se encuentra activado cuando se dispone de datos no leídos en el buffer de recepción, y desactivado cuando dicho buffer se encuentra vacío. Si el Receptor es deshabilitado (RXENn = 0) el contenido del buffer de recepción será borrado, y en consecuencia el *flag* RXCn se desactivará.

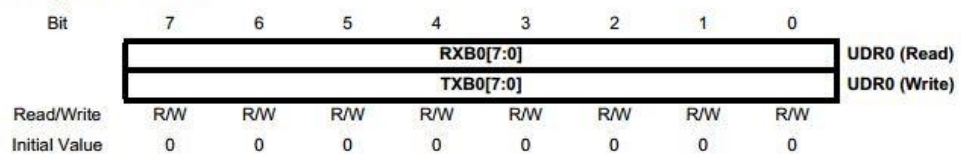
Cuando el bit de habilitación de la interrupción por recepción completada (*Receive Complete Interrupt Enable, TXCIEn*) en el registro UCSRnB se pone a “1” lógico, se ejecutará dicha interrupción siempre que el *flag* TXCn se active, asumiendo que las interrupciones globales están habilitadas. Cuando la recepción es gestionada mediante interrupciones, la rutina de servicio debe leer los datos recibidos en UDRn con el fin de desactivar el *flag* RXCn, de lo contrario se volverá a generar una nueva interrupción una vez finalice la rutina de servicio.

Al contrario del Transmisor, deshabilitar el Receptor tiene efecto inmediatamente. Por tanto se perderán los datos de cualquier recepción en curso. Una vez deshabilitado, el Receptor no continuará sobrescribiendo la función del pin RxDn y los datos que queden en el buffer de recepción se perderán.

### 2.4.2.5 Descripción de registros.

En este apartado se explicarán cada uno de los registros utilizados en la configuración y la gestión de la USART, así como la descripción del significado de los bits dentro de cada registro. Únicamente se explicarán los registros y bits relevantes a este TFG. En primer lugar se pueden observar los registros de datos en la Figura 2.17.

#### USART0 I/O Data Register – UDR0



#### USART1 I/O Data Register – UDR1

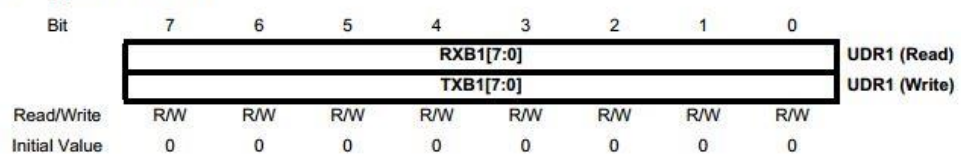


Figura 2.17. Registro de datos I/O de la USART

## Descripción Hardware

- **Bit 7:0 – RxBn7:0: Receive Data Buffer.**
- **Bit 7:0 – TxBn7:0: Transmit Data Buffer.**

Los buffers de recepción y transmisión de la USART comparten la misma dirección en el espacio de memoria, referido como *USARTn Data Register* o UDRn. El buffer de transmisión (TXBn) será el destino de los datos escritos en la dirección del registro UDRn, mientras que leer del registro UDRn devolverá el contenido del buffer de recepción (RXBn).

En las figuras que siguen a continuación se pueden observar los registros de estado y de control de la USART.

### USART0 Control and Status Register A – UCSR0A

Bit	7	6	5	4	3	2	1	0	
	<b>RXC0</b>	<b>TXC0</b>	<b>UDRE0</b>	<b>FE0</b>	<b>DOR0</b>	<b>UPE0</b>	<b>U2X0</b>	<b>MPCM0</b>	UCSR0A
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

### USART1 Control and Status Register A – UCSR1A

Bit	7	6	5	4	3	2	1	0	
	<b>RXC1</b>	<b>TXC1</b>	<b>UDRE1</b>	<b>FE1</b>	<b>DOR1</b>	<b>UPE1</b>	<b>U2X1</b>	<b>MPCM1</b>	UCSR1A
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Figura 2.18. Registro de Estado y de Control A

- **Bit 7 – RXCn: USARTn Receive Complete.**

Este bit se activa cuando hay datos en el buffer de recepción que no han sido leídos, y se desactiva cuando el buffer de recepción se vacía. Si el Receptor se deshabilita, los datos que hayan en el buffer de recepción son descartados y por tanto el *flag* RXCn se desactiva. Este *flag* puede ser utilizado para generar una interrupción.

- **Bit 6 – TXCn: USARTn Transmit Complete.**

Este bit se activa cuando la trama completa del Registro de Desplazamiento se ha desplazado hacia la salida y no hay nuevos datos en el buffer de transmisión. El *flag* TXCn



se desactiva automáticamente cuando la interrupción de transmisión completada se ejecuta, o bien escribiendo un “1” lógico en el bit correspondiente al *flag*. Este *flag* puede ser utilizado para generar una interrupción.

**USART0 Control and Status Register B – UCSR0B**

Bit	7	6	5	4	3	2	1	0	
	<b>RXCIE0</b>	<b>TXCIE0</b>	<b>UDRIE0</b>	<b>RXEN0</b>	<b>TXEN0</b>	<b>UCSZ02</b>	<b>RXB80</b>	<b>TXB80</b>	<b>UCSR0B</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

**USART1 Control and Status Register B – UCSR1B**

Bit	7	6	5	4	3	2	1	0	
	<b>RXCIE1</b>	<b>TXCIE1</b>	<b>UDRIE1</b>	<b>RXEN1</b>	<b>TXEN1</b>	<b>UCSZ12</b>	<b>RXB81</b>	<b>TXB81</b>	<b>UCSR1B</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.19. Registro de Estado y de Control B

- **Bit 7 – RXCIEn: RX Complete Interrupt Enable.**

Escribir un “1” lógico en este bit habilita la interrupción del *flag* RXCn. Se generará una interrupción de recepción completada sólo si el bit RXCIEn está a nivel alto, las interrupciones globales están habilitadas y el *flag* RXCn se encuentra activo.

- **Bit 6 – TXCIEn: TX Complete Interrupt Enable.**

Escribir un “1” lógico en este bit habilita la interrupción del *flag* TXCn. Se generará una interrupción de transmisión completada sólo si el bit TXCIEn está a nivel alto, las interrupciones globales están habilitadas y el *flag* TXCn se encuentra activo.

- **Bit 4 – RXENn: Receiver Enable.**

Escribir un “1” lógico en este bit habilita el Receptor de la USART. El Receptor ignorará el modo de operación normal del pin RxDn cuando está habilitado. Deshabilitar el Receptor descartará el contenido del buffer de recepción.

- **Bit 3 – TXENn: *Transmitter Enable*.**

Escribir un “1” lógico en este bit habilita el Transmisor de la USART. El Transmisor ignorará el modo de operación normal del pin TxDn cuando está habilitado. Deshabilitar el Transmisor no se hará efectivo hasta que se completen todas las transmisiones pendientes.

- **Bit 2 – UCSZn2: *Character Size*.**

Los bits UCSZn2 combinados con los bits UCSZn1:0 en el registro UCSRnC establecen el número de bits de datos para las tramas, tanto del Receptor como del Transmisor.

**USART0 Control and Status Register C – UCSR0C**

Bit	7	6	5	4	3	2	1	0	
	-	UMSEL0	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0	UCSR0C
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

**USART1 Control and Status Register C – UCSR1C**

Bit	7	6	5	4	3	2	1	0	
	-	UMSEL1	UPM11	UPM10	USBS1	UCSZ11	UCSZ10	UCPOL1	UCSR1C
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

Figura 2.20. Registro de Estado y de Control C

- **Bit 6 – UMSELn: *USARTn Mode Select*.**

Este bit permite seleccionar entre los modos de operación síncrono y asíncrono.

UMSELn	Mode
0	Asynchronous Operation
1	Synchronous Operation

Figura 2.21. Configuración del bit UMSELn

- **Bit 5:4 – UPMn1:0: Parity Mode.**

Estos bits habilitan y establecen el tipo de generación y chequeo de paridad. Si está habilitada, el Transmisor automáticamente generará y enviará el bit de paridad correspondiente a los bits de datos enviados en cada trama. El Receptor por su parte genera la paridad de los datos recibidos y lo compara con el valor establecido en UPMn.

UPMn1	UPMn0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

*Figura 2.22. Configuración de los bits UPMn*

- **Bit 3 – USBSn: Stop Bit Select.**

Este bit selecciona el número de bits de *stop* a ser insertados por el Transmisor. El Receptor ignora esta configuración, ya que solamente utiliza el primer bit de *stop*.

USBSn	Stop Bit(s)
0	1-bit
1	2-bit

*Figura 2.23. Configuración del bit USBSn*

- **Bit 2:1 – UCSZn1:0: Character Size.**

Los bits UCSZn1:0, combinados con el bit UCSZn2 en el registro UCSRnB, establecen el número de bits de datos para las tramas, tanto del Receptor como del Transmisor.

UCSZn2	UCSZn1	UCSZn0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Figura 2.24. Configuración de los bits UCSZn

### 2.4.3 CAN.

El protocolo CAN (*Controller Area Network*) es un protocolo de tipo *broadcast*, serie y de tiempo real con un alto nivel de seguridad. El controlador CAN del AT90CAN128 es completamente compatible con la Especificación CAN 2.0 en sus versiones A y B [8]. Además proporciona las características necesarias para implementar el protocolo CAN de acuerdo con el modelo de referencia OSI de la ISO:

Capa de Enlace de Datos:

- Subcapa de control de enlace lógico (*Logical Link Control, LLC*).
- Subcapa de control de acceso al medio (*Medium Access Control, MAC*).

Capa Física:

- Subcapa de señalización física (*Physical Signalling, PLS*).

Este controlador CAN es capaz de gestionar todo tipo de tramas (*Data, Remote, Error y Overload*), además de una serie de características que se mencionan a continuación:

- Controlador CAN completo.
- Completamente compatible con el Estándar CAN rev 2.0 A y rev 2.0 B.
- 15 MOb (*Message Object*) con sus propios:

- 11 bits de identificador (rev 2.0 A) y 29 bits de identificador (rev 2.0 B).
- 11 bits de máscara (rev 2.0 A) y 29 bits de máscara (rev 2.0 B).
- Buffer de datos de 8 bytes.
- Configuraciones de Tx, Rx, Buffer y Respuesta Automática.
- Máxima tasa de transferencia de 1Mbit/s a 8MHz.

#### 2.4.3.1 Protocolo CAN.

El protocolo CAN es un estándar internacional definido en el ISO 11898 para alta velocidad y en el ISO 11519-2 para baja velocidad. CAN está basado en un mecanismo de comunicación *broadcast*. Esta comunicación *broadcast* se alcanza usando un protocolo de transmisión orientado a mensaje. Estos mensajes son identificados por un identificador, el cual debe ser único dentro de toda la red, y define, no sólo el contenido, sino también la prioridad del mensaje [12, 13].

La prioridad con la que se transmite un mensaje comparado con otro mensaje de menor prioridad se especifica mediante el identificador. Las prioridades se definen durante el diseño del sistema en forma de los correspondientes valores binarios y no puede ser cambiada dinámicamente. El identificador con el número binario más bajo posee la prioridad más alta.

Los conflictos en el acceso al bus se resuelven mediante un arbitraje bit a bit de los identificadores involucrados. Esto sucede de acuerdo con el mecanismo de AND cableada, a través del cual el estado dominante anula al estado recesivo. Todos los nodos con una transmisión recesiva pierden la competencia por el reparto del bus. Los perdedores automáticamente se convierten en receptores del mensaje con la prioridad más alta, y no vuelven a intentar transmitir hasta que el bus haya sido liberado.

## Descripción Hardware

El protocolo CAN soporta dos formatos de trama de los mensajes, que se diferencian básicamente en la longitud del identificador. La trama CAN estándar, también conocida como CAN 2.0 A, soporta una longitud de 11 bits para el identificador, mientras que la trama CAN extendida, también conocida como CAN 2.0 B, soporta una longitud de 29 bits para el identificador.

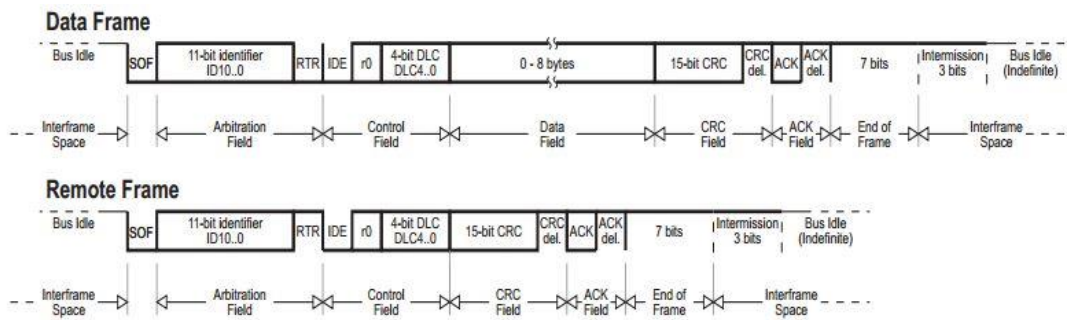


Figura 2.25. Trama CAN estándar

Un mensaje en el formato CAN estándar comienza con el “inicio de trama” (*Start Of Frame*, SOF). A esto le sigue el “campo de arbitraje” (*Arbitration Field*) que consiste en el identificador y el bit de “petición de transmisión remota” (*Remote Transmission Request*, RTR), utilizado para distinguir entre una trama de datos y una trama de petición de datos, llamada trama remota. A continuación está el “campo de control” (*Control Field*) el cual contiene el bit de “extensión de identificador” (*IDentifier Extension*, IDE) y el “código de longitud de datos” (*Data Length Code*, DLC), que se usa para indicar el número de bytes que siguen a continuación en el “campo de datos” (*Data Field*), el cual puede contener hasta un máximo de 8 bytes. La integridad de la trama se garantiza con el “código de redundancia cíclica” (*Cyclic Redundant Check*, CRC) que le sigue. El “campo de reconocimiento” (*ACKnowledge Field*, ACK) está compuesto por el bit de ACK y el delimitador de ACK. El bit de ACK se envía como un bit recesivo y es sobrescrito como un bit dominante por los receptores que hayan recibido la trama correctamente. Los mensajes correctos son reconocidos por los receptores sin importar la configuración del filtro de

aceptación de los mismos. El fin del mensaje es indicado por el “fin de trama” (*End Of Frame, EOF*). El “espacio de intervalo entre tramas” (*Intermission Frame Space, IFS*) es el número de bits mínimo que separa dos mensajes consecutivos. Si a continuación ningún nodo accede al bus, éste permanece inactivo.

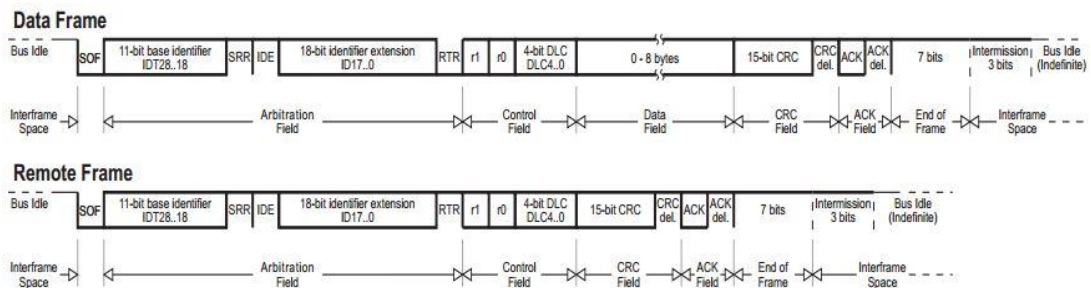


Figura 2.26. Trama CAN extendida

Un mensaje en el formato CAN extendido es prácticamente igual al formato CAN estándar. La diferencia está en la longitud del identificador. El identificador se compone del identificador de 11 bits existente (*base identifier*) y una extensión de 18 bits (*identifier extension*). La distinción entre el formato estándar y el extendido se hace a través del bit IDE, el cual se transmite como dominante en caso de tratarse de una trama CAN estándar, y como recesivo en el otro caso.

Como los dos formatos tienen que coexistir en un mismo bus, se establece el mensaje que tiene la prioridad más alta en caso de colisión en el acceso al bus de dos formatos diferentes con el mismo identificador: El mensaje en formato CAN estándar siempre tiene prioridad sobre los mensajes en formato extendido.

Durante la transmisión, el arbitraje del bus se puede perder en favor de otro dispositivo con un identificador de mayor prioridad. Este concepto de arbitraje evita las colisiones de mensajes cuya transmisión comenzó de forma simultánea en varios nodos.

## Descripción Hardware

Por lo general, los conflictos de acceso al bus se resuelven en el campo de arbitraje a partir del valor del identificador. Si una trama de datos y una trama remota con el mismo identificador se envían al mismo tiempo, la trama de datos prevalece sobre la trama remota.

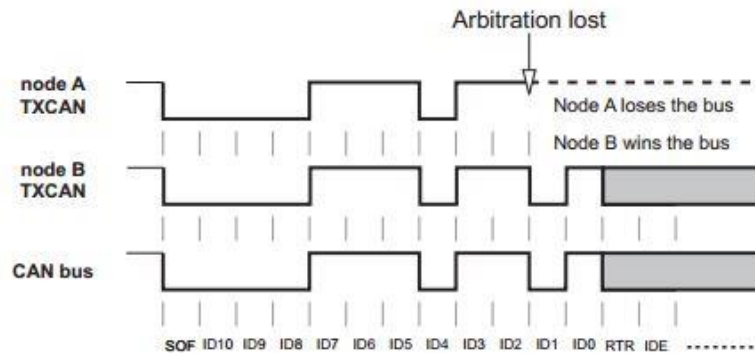


Figura 2.27. Arbitraje del bus

### 2.4.3.2 Controlador CAN.

El controlador CAN implementado en el AT90CAN128 soporta la versión 2.0 B del estándar CAN. Este controlador CAN completo proporciona todo el hardware necesario para un filtrado de aceptación adecuado y la gestión de mensajes. Para cada mensaje a ser transmitido o recibido, este módulo contiene lo que se llama un “objeto de mensaje” (*Message Object, MOB*), en el cual se almacena toda la información relacionada con el mensaje (identificador, bytes de datos, etc.).

Durante la inicialización del periférico, la aplicación define qué MOBs son para enviar y cuales para recibir. Sólo si el controlador CAN recibe un mensaje cuyo identificador coincide con uno de los identificadores programados en los MOBs destinados a la recepción se guarda el mensaje y la aplicación es informada mediante una interrupción. Otra ventaja es que las tramas remotas entrantes pueden ser respondidas automáticamente por el controlador con la correspondiente trama de datos. De esta forma, la carga de la CPU se



reduce considerablemente en comparación con una solución CAN básica. Usando este controlador CAN completo, se pueden gestionar altas tasas de baudios y buses cargados con muchos mensajes.

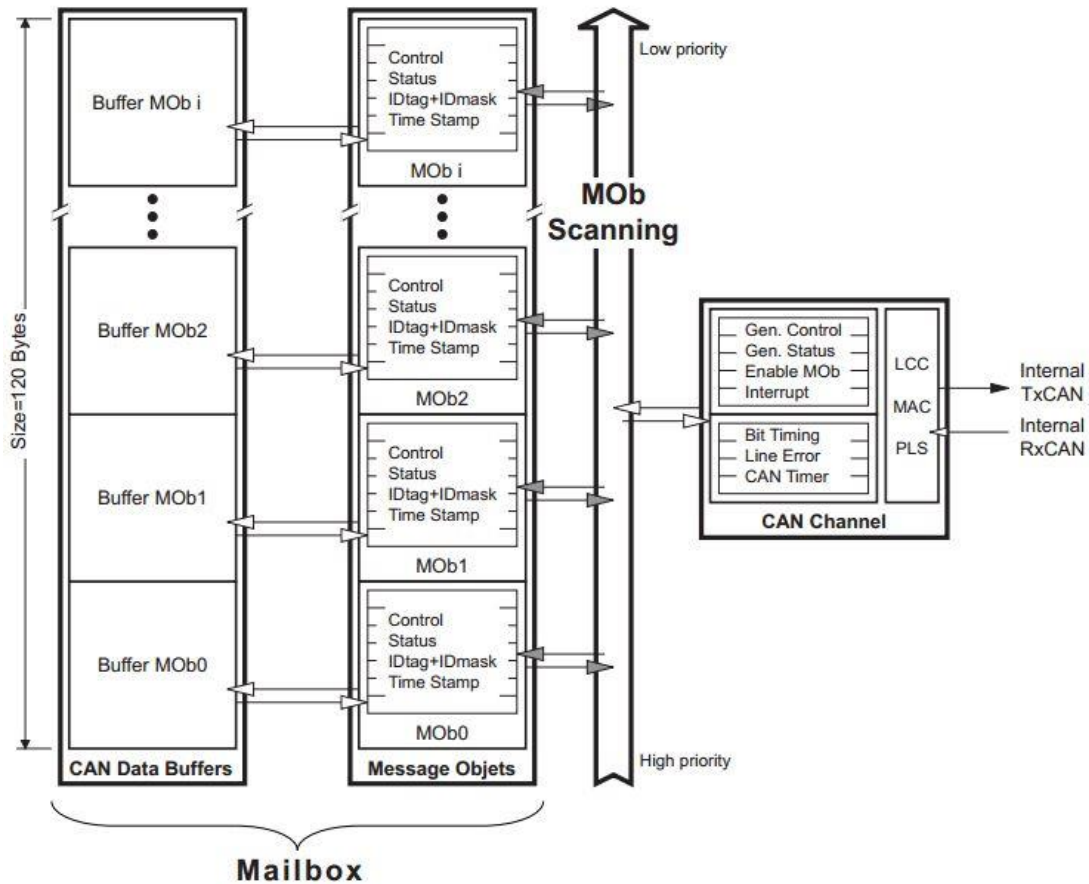


Figura 2.28. Estructura del controlador CAN

### 2.4.3.3 Objetos de mensajes (MOB).

Un MOB describe una trama CAN, conteniendo toda la información para controlar una trama. Esto quiere decir que los MOBs han sido definidos de forma que permitan describir un mensaje CAN como si fuera un objeto. Este dispositivo tiene 15 MOBs, numerados del 0 al 14. Los MOBs son independientes entre sí pero se le da prioridad al de menor número en caso de que se produzcan múltiples coincidencias. Cada MOB tiene su

## Descripción Hardware

propio campo para controlar el modo de operación. Antes de habilitar el periférico CAN, cada MOB debe ser configurado, ya que no existe un modo de operación por defecto. Los modos de operación se pueden observar en la Figura 2.29.

MOB Configuration		Reply Valid	RTR Tag	Operating Mode
0	0	x	x	Disabled
0	1	x	0	Tx Data Frame
		x	1	Tx Remote Frame
1	0	x	0	Rx Data Frame
		0	1	Rx Remote Frame
		1		Rx Remote Frame then, Tx Data Frame (reply)
1	1	x	x	Frame Buffer Receive Mode

Figura 2.29. Configuración de los MOBs

Cada MOB está mapeado en una página para ahorrar espacio. El número de página se corresponde con el número de MOB, el cual se establece en el registro CANPAGE. El registro CANHPMOB devuelve el MOB con mayor prioridad que haya generado una interrupción en el registro CANSIT.

Cada MOB contiene un buffer para los datos, el cual puede ser accedido a través del registro CANMSG. El índice de datos (INDX) es el puntero de dirección al byte de datos requerido. El byte de datos puede ser leído o escrito. El índice de datos se autoincrementa después de cada acceso siempre que el bit AINC en el registro CANPAGE esté a nivel bajo. Además, se ha implementado un reinicio del índice, con lo que después de INDX=7 se pasa a INDX=0.

Para enviar una trama se sigue el siguiente proceso:

- Se deben inicializar varios campos antes de poder enviar:

- Identificador (IDT).
- Extensión de identificador (IDE).
- Petición de transmisión remota (RTRTAG).
- Código de longitud de datos (DLC).
- Bytes de datos del mensaje (MSG).
- El MOB estará listo para enviar la trama cuando se configure como transmisor en el registro CONMOB.
- Entonces se escanean todos los MOBs configurados como transmisor, se encuentra el de mayor prioridad y se intenta enviar.
- Cuando la transmisión se ha completado se activa el *flag* TXOK.
- Todos los parámetros y datos están disponibles en el MOB hasta una nueva inicialización.

Por otro lado, el proceso para la recepción es el siguiente:

- Se deben inicializar varios campos antes de poder recibir:
  - Identificador (IDT).
  - Máscara de identificador (IDMSK).
  - Extensión de identificador (IDE).
  - Máscara de extensión de identificador (IDEMSK).
  - Petición de transmisión remota (RTRTAG).
  - Máscara de petición de transmisión remota (RTRMSK).
  - Código de longitud de datos (DLC).
- El MOB estará listo para recibir cuando se configure como receptor en el registro CONMOB.
- Cuando se recibe un identificador de trama, se escanean todos los MOBs en modo de recepción y se intenta encontrar el MOB con mayor prioridad que genere una coincidencia.

## Descripción Hardware

- Ante una coincidencia, el IDT, IDE y DLC del MOB que ha generado la coincidencia se actualizan con los valores de la trama recibida.
- Una vez se ha completado la recepción, los bytes de datos recibidos se almacenan en el buffer de datos del MOB que ha generado la coincidencia y se activa el *flag* RXOK.
- Todos los parámetros y datos del MOB están disponibles hasta una nueva inicialización.

### 2.4.3.4 Interrupciones.

Cuando ocurre una interrupción, un bit que actúa como *flag* se activa en el registro CANSTMOB del MOB correspondiente, o en el registro general CANGIT. Si los bits ENRX, ENTX o ENERR en el registro CANIE están a “1” lógico, entonces se activa el bit correspondiente al MOB en el registro CANSITn.

Para reconocer una interrupción generada por un MOB, los bits correspondientes (RXOK, TXOK,...) en el registro CANSTMOB deben ser limpiados por la aplicación. Para reconocer una interrupción general, los bits correspondientes (BXOK, BOFFIT,...) en el registro CANGIT deben ser inicializados por la aplicación. Esta operación se realiza escribiendo un “1” lógico en estos *flags* de interrupción (escribir un “0” lógico no modifica el valor de los mismos).

### 2.4.3.5 Descripción de los registros generales.

En este apartado se explicarán cada uno de los registros generales utilizados en la configuración y la gestión del CAN, así como la descripción del significado de los bits dentro de cada registro. Únicamente se explicarán aquellos que sean relevantes a este TFG. En primer lugar se puede observar el registro de control general en la Figura 2.30.

Bit	7	6	5	4	3	2	1	0	
	<b>ABRQ</b>	<b>OVRQ</b>	<b>TTC</b>	<b>SYNTTC</b>	<b>LISTEN</b>	<b>TEST</b>	<b>ENA/STB</b>	<b>SWRES</b>	<b>CANGCON</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.30. Registro de control general

- **Bit 1 – ENA/STB: Enable / Standby Mode.**

Debido a que este bit es un comando y no es efectivo inmediatamente, el bit ENFG en el registro CANGSTA indica el verdadero estado del controlador.

- 0: modo *standby*: Si existe alguna transmisión en curso se termina normalmente y el CAN se detiene. El transmisor constantemente establece un nivel recesivo. En este modo, el receptor no está habilitado pero todos los registros y MObs continúan siendo accesibles por la CPU.

- 1: modo habilitado: El CAN se habilita una vez se hayan leído 11 bits recesivos.

- **Bit 0 – SWRES: Software Reset Request.**

Este bit inicializa el controlador CAN.

- 0: no *reset*.

- 1: *reset*.

Bit	7	6	5	4	3	2	1	0	
	<b>CANIT</b>	<b>BOFFIT</b>	<b>OVRTIM</b>	<b>BXOK</b>	<b>SERG</b>	<b>CERG</b>	<b>FERG</b>	<b>AERG</b>	<b>CANGIT</b>
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.31. Registro de interrupciones generales

## Descripción Hardware

- **Bit 7 – CANIT: *General Interrupt Flag.***

Este es un bit de sólo lectura.

- 0: no hay interrupciones.

- 1: interrupción CAN: es una imagen de todas las interrupciones del controlador CAN. Este bit puede ser usado para métodos de acceso basados en *polling*.

Bit	7	6	5	4	3	2	1	0	CANGIE
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.32. Registro de habilitación de interrupciones generales

- **Bit 7 – ENIT: *Enable all Interrupts.***

- 0: interrupciones deshabilitadas.

- 1: interrupciones habilitadas (CANIT).

- **Bit 5 – ENRX: *Enable Receive Interrupt.***

- 0: interrupción deshabilitada.

- 1: interrupciones por recepción habilitadas.

- **Bit 4 – ENTX: *Enable Transmit Interrupt.***

- 0: interrupción deshabilitada.

- 1: interrupciones por transmisión habilitadas.

Bit	7	6	5	4	3	2	1	0	
	ENMOB7	ENMOB6	ENMOB5	ENMOB4	ENMOB3	ENMOB2	ENMOB1	ENMOB0	CANEN2
	-	ENMOB14	ENMOB13	ENMOB12	ENMOB11	ENMOB10	ENMOB9	ENMOB8	CANEN1
Bit	15	14	13	12	11	10	9	8	
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
Read/Write	-	R	R	R	R	R	R	R	
Initial Value	-	0	0	0	0	0	0	0	

Figura 2.33. Registros de habilitación de los MOBs

- **Bits 14:0 - ENMOB14:0: Enable MOB.**

Este bit indica la disponibilidad del MOB.

- 0: MOB deshabilitado, disponible para una nueva transmisión o recepción.
- 1: MOB habilitado, en uso.

Bit	7	6	5	4	3	2	1	0	
	IEMOB7	IEMOB6	IEMOB5	IEMOB4	IEMOB3	IEMOB2	IEMOB1	IEMOB0	CANIE2
	-	IEMOB14	IEMOB13	IEMOB12	IEMOB11	IEMOB10	IEMOB9	IEMOB8	CANIE1
Bit	15	14	13	12	11	10	9	8	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
Read/Write	-	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	0	0	0	0	0	0	0	

Figura 2.34. Registros de habilitación de las interrupciones de los MOBs

- **Bits 14:0 - IEMOB14:0: Interrupt Enable by MOB.**

- 0: interrupciones deshabilitadas.
- 1: interrupciones de los MOB habilitadas.

Ejemplo: CANIE2 = 0000 1100: habilita las interrupciones de los MOBs 2 y 3.

## Descripción Hardware

Bit	7	6	5	4	3	2	1	0	
	<b>MOBNB3</b>	<b>MOBNB2</b>	<b>MOBNB1</b>	<b>MOBNB0</b>	<b>A<math>\bar{I}</math>NC</b>	<b>INDX2</b>	<b>INDX1</b>	<b>INDX0</b>	<b>CANPAGE</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.35. Registro de página MOB

- **Bit 7:4 – MOBNB3:0: MOb Number.**

Selecciona el número de MOB, yendo los números disponibles desde el 0 al 14.

- **Bit 3 – AINC: Auto Increment of the FIFO CAN Data Buffer Index.**

- 0: autoincremento del índice activado (por defecto).

- 1: autoincremento del índice desactivado.

- **Bit 2:0 – INDX2:0: FIFO CAN Data Buffer Index.**

Ubicación del byte de datos en la FIFO para el MOB definido.

### 2.4.3.6 Descripción de los registros MOB.

En este apartado se explicarán cada uno de los registros relacionados con los MOBs que han sido utilizados en la configuración y la gestión del CAN, así como la descripción del significado de los bits dentro de cada registro. Únicamente se explicarán aquellos que sean relevantes a este TFG. En primer lugar se puede observar el registro de estado en la Figura 2.36.



Bit	7	6	5	4	3	2	1	0	CANSTMOB
	DLCW	TXOK	RXOK	BERR	SERR	CERR	FERR	AERR	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

Figura 2.36. Registro de estado del MOB

- **Bit 6 – TXOK: *Transmit OK*.**

Este *flag* puede generar una interrupción. Indica una transmisión completada.

- **Bit 5 – RXOK: *Receive OK*.**

Este *flag* puede generar una interrupción. Indica una recepción completada.

Bit	7	6	5	4	3	2	1	0	CANCDMOB
	CONMOB1	CONMOB0	RPLV	IDE	DLC3	DLC2	DLC1	DLC0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

Figura 2.37. Registro de control y DLC del MOB

- **Bit 7:6 – CONMOB1:0: *Configuration of Message Object*.**

Estos bits establecen el tipo de comunicación que se va a realizar (sin valor inicial después de un *reset*). Estos bits no se inicializan una vez se ha realizado la comunicación. El usuario debe reescribir la configuración para iniciar una nueva comunicación.

- 00: deshabilitado.
- 01: transmisión.
- 10: recepción.
- 11: recepción en buffer.

Descripción Hardware

- **Bit 4 – IDE: Identifier Extension.**

Bit de extensión de identificador (IDE) de la trama a enviar. Este bit se actualiza con el valor correspondiente de la trama recibida.

- 0: Estándar CAN rev 2.0 A (longitud de identificador de 11 bits).

- 1: Estándar CAN rev 2.0 B (longitud de identificador de 29 bits).

- **Bit 3:0 – DLC3:0: Data Length Code.**

Número de bytes en el campo de datos del mensaje. Equivale al campo DLC de la trama a enviar. El rango permitido es de 0 a 8 bytes, si se supera esta cantidad, entonces el valor efectivo es DLC = 8. Este campo se actualiza con el valor correspondiente de la trama recibida.

*V2.0 part A*

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	-	-	-	-	-	RTRTAG	-	RB0TAG	CANIDT4
	-	-	-	-	-	-	-	-	CANIDT3
	IDT2	IDT1	IDT0	-	-	-	-	-	CANIDT2
	IDT10	IDT9	IDT8	IDT7	IDT6	IDT5	IDT4	IDT3	CANIDT1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

*V2.0 part B*

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	IDT4	IDT3	IDT2	IDT1	IDT0	RTRTAG	RB1TAG	RB0TAG	CANIDT4
	IDT12	IDT11	IDT10	IDT9	IDT8	IDT7	IDT6	IDT5	CANIDT3
	IDT20	IDT19	IDT18	IDT17	IDT16	IDT15	IDT14	IDT13	CANIDT2
	IDT28	IDT27	IDT26	IDT25	IDT24	IDT23	IDT22	IDT21	CANIDT1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

Figura 2.38. Registro de identificador

- **Bit 31:3 – IDT28:0: *Identifier Tag.***

Identificador de la trama a enviar. Este campo se actualiza con el valor correspondiente de la trama recibida.

- **Bit 2 – RTRTAG: *Remote Transmission Request Tag.***

Bit RTR de la trama a enviar. Este bit se actualiza con el valor correspondiente de la trama recibida.

- **Bit 1 – RB1TAG: *Reserved Bit 1 Tag.***

Bit RB1 de la trama a enviar. Este bit se actualiza con el valor correspondiente de la trama recibida.

- **Bit 0 – RB0TAG: *Reserved Bit 0 Tag.***

Bit RB0 de la trama a enviar. Este bit se actualiza con el valor correspondiente de la trama recibida.

## Descripción Hardware

### V2.0 part A

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	-	-	-	-	-	RTRMSK	-	IDEMSK	CANIDM4
	-	-	-	-	-	-	-	-	CANIDM3
	IDMSK2	IDMSK1	IDMSK0	-	-	-	-	-	CANIDM2
	IDMSK10	IDMSK9	IDMSK8	IDMSK7	IDMSK6	IDMSK5	IDMSK4	IDMSK3	CANIDM1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

### V2.0 part B

Bit	15/7	14/6	13/5	12/4	11/3	10/2	9/1	8/0	
	IDMSK4	IDMSK3	IDMSK2	IDMSK1	IDMSK0	RTRMSK	-	IDEMSK	CANIDM4
	IDMSK12	IDMSK11	IDMSK10	IDMSK9	IDMSK8	IDMSK7	IDMSK6	IDMSK5	CANIDM3
	IDMSK20	IDMSK19	IDMSK18	IDMSK17	IDMSK16	IDMSK15	IDMSK14	IDMSK13	CANIDM2
	IDMSK28	IDMSK27	IDMSK26	IDMSK25	IDMSK24	IDMSK23	IDMSK22	IDMSK21	CANIDM1
Bit	31/23	30/22	29/21	28/20	27/19	26/18	25/17	24/16	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

Figura 2.39. Registro de máscara del identificador

- **Bit 31:3 – IDMSK28:0: Identifier Mask.**

- 0: Comparación ignorada.
- 1: Comparación de bit habilitada.

- **Bit 2 – RTRMSK: Remote Transmission Request Mask.**

- 0: Comparación ignorada.
- 1: Comparación de bit habilitada.

- **Bit 0 – IDEMSK: Identifier Extension Mask.**

- 0: Comparación ignorada.
- 1: Comparación de bit habilitada.

A continuación en la Figura 2.40 se puede observar un ejemplo del uso de la máscara de identificador.

**Full filtering:** to accept only ID = 0x317 in part A.  
 - ID MSK = 111 1111 1111<sub>b</sub>  
 - ID TAG = 011 0001 0111<sub>b</sub>

**Partial filtering:** to accept ID from 0x310 up to 0x317 in part A.  
 - ID MSK = 111 1111 1000<sub>b</sub>  
 - ID TAG = 011 0001 0xxx<sub>b</sub>

**No filtering:** to accept all ID from 0x000 up to 0x7FF in part A.  
 - ID MSK = 000 0000 0000<sub>b</sub>  
 - ID TAG = xxx xxxxx xxxxx<sub>b</sub>

Figura 2.40. Ejemplo de máscara

Bit	7	6	5	4	3	2	1	0	
	MSG 7	MSG 6	MSG 5	MSG 4	MSG 3	MSG 2	MSG 1	MSG 0	CANMSG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	-	-	-	-	-	-	-	-	

Figura 2.41. Registro de datos del mensaje

- **Bit 7:0 – MSG7:0: Message Data.**

Este registro contiene el byte de datos apuntado en el registro de la página del MOB. Cuando se escribe en el registro de la página del MOB, este byte es igual al mensaje localizado en la dirección del identificador base del MOB, más el índice. Si se está utilizando el autoincremento, al terminar una lectura o escritura de este registro, se autoincrementa el índice.

## 2.4.4 Bootloader.

El soporte de *Bootloader* proporciona al microcontrolador (MCU) un mecanismo de autoprogramación que le permite recibir y cargar código de programa en la memoria por

sí mismo. Esta característica permite mayor flexibilidad en las actualizaciones de software controladas por el MCU usando un programa *bootloader* residente en memoria *Flash*. El programa *bootloader* puede utilizar cualquier interfaz de datos y protocolo asociado que esté disponible para leer el código y escribirlo en la memoria *Flash*. El programa residente en la sección de *bootloader* tiene la capacidad de escribir en toda la memoria *Flash*, incluyendo la propia sección de *bootloader*. Por lo tanto puede incluso modificarse a sí mismo, o borrarse en caso de que no se necesite más esa característica. El tamaño de la zona de *bootloader* es configurable a través de unos fusibles [8].

La memoria *Flash* está organizada en dos secciones principales, la sección de Aplicación (*Application Section, AS*) y la sección de *Bootloader (Bootloader Section, BLS)*. El tamaño de las distintas secciones se configura a través de los fusibles *BOOTSZ*. La sección de Aplicación es la zona de la memoria *Flash* que se utiliza para almacenar el código de aplicación. Esta sección nunca puede contener código de *bootloader* puesto que la instrucción *SPM (Store Program Memory)* está deshabilitada dentro de esta sección. Por otro lado, el software de *bootloader* debe ser almacenado en la sección *BLS* debido a que la instrucción *SPM* puede iniciar la programación sólo cuando se ejecuta desde esta sección. La instrucción *SPM* puede acceder a toda la memoria *Flash*, incluyendo la propia sección *BLS*. En la Figura 2.42 se pueden observar las distintas combinaciones de estas secciones dependiendo de la configuración de los fusibles.

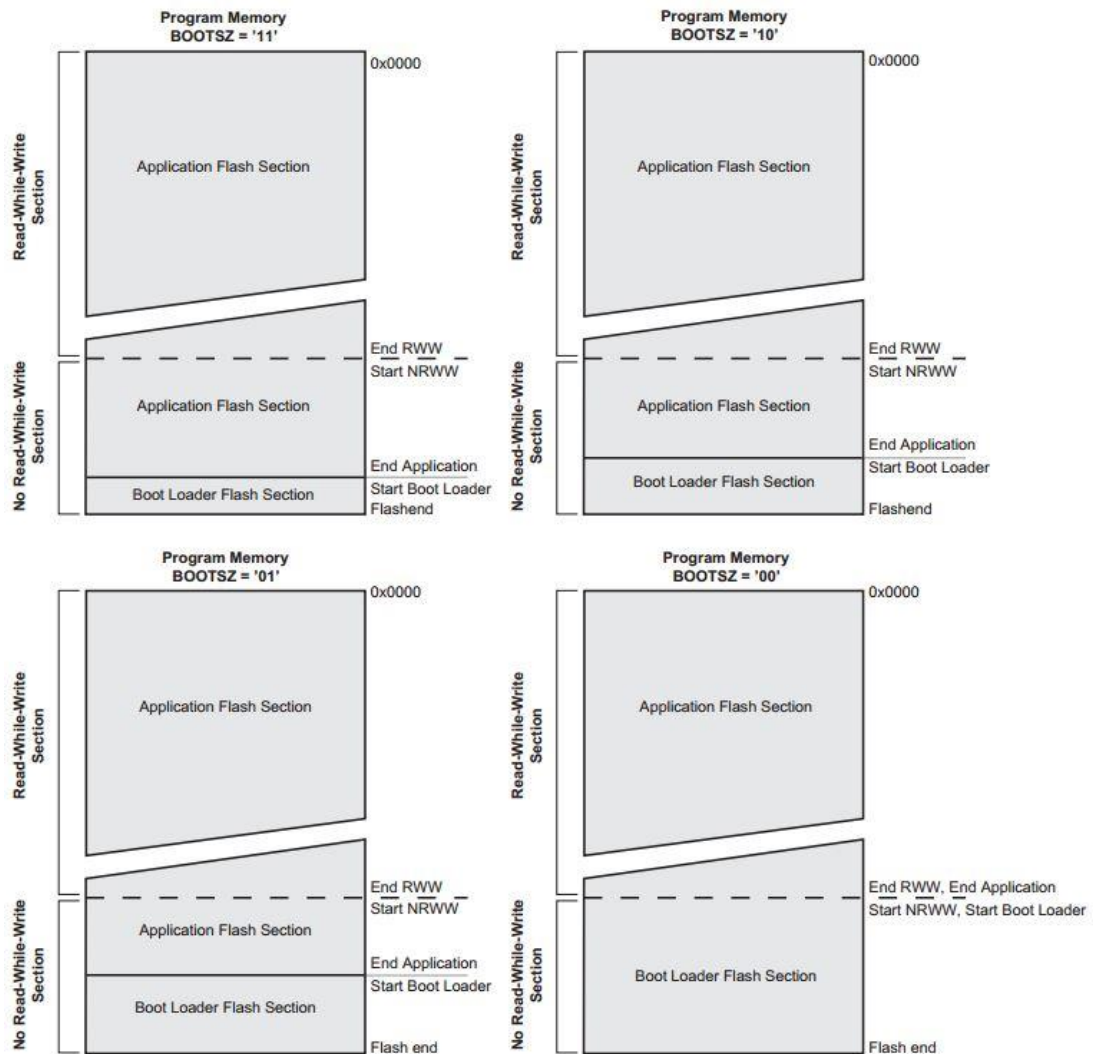


Figura 2.42. Secciones de la memoria Flash

El puntero Z se utiliza para direccionar los comandos SPM. Este puntero está compuesto por los registros ZH, ZL y RAMPZ. El número de bits realmente utilizados es dependiente de la implementación. El registro RAMPZ sólo se utiliza cuando el espacio de memoria a programar es superior a los 64KBytes. Una vez la operación de programar la memoria se inicia, el puntero Z se pasa por un *latch* y puede ser utilizado para otras operaciones.

## Descripción Hardware

Bit	23	22	21	20	19	18	17	16
	15	14	13	12	11	10	9	8
RAMPZ	-	-	-	-	-	-	-	RAMPZ0
ZH (R31)	Z15	Z14	Z13	Z12	Z11	Z10	Z9	Z8
ZL (R30)	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0
	7	6	5	4	3	2	1	0

Figura 2.43. Puntero Z

Dado que la memoria *Flash* está organizada en páginas, las cuales se pueden observar en la Figura 2.44, el contador de programa (*Program Counter, PC*) se puede tratar como si tuviera dos secciones diferentes. Una sección, que consiste en los bits menos significativos, direcciona las palabras dentro de una página, mientras que los bits más significativos direccionan las páginas.

Device	Flash Size	Page Size	PCWORD	No. of Pages	PCPAGE	PCMSB
AT90CAN32	16K words	128 words	PC[6:0]	128	PC[13:7]	13
AT90CAN64	32K words	128 words	PC[6:0]	256	PC[14:7]	14
AT90CAN128	64K words	128 words	PC[6:0]	512	PC[15:7]	15

Figura 2.44. Páginas de la memoria Flash



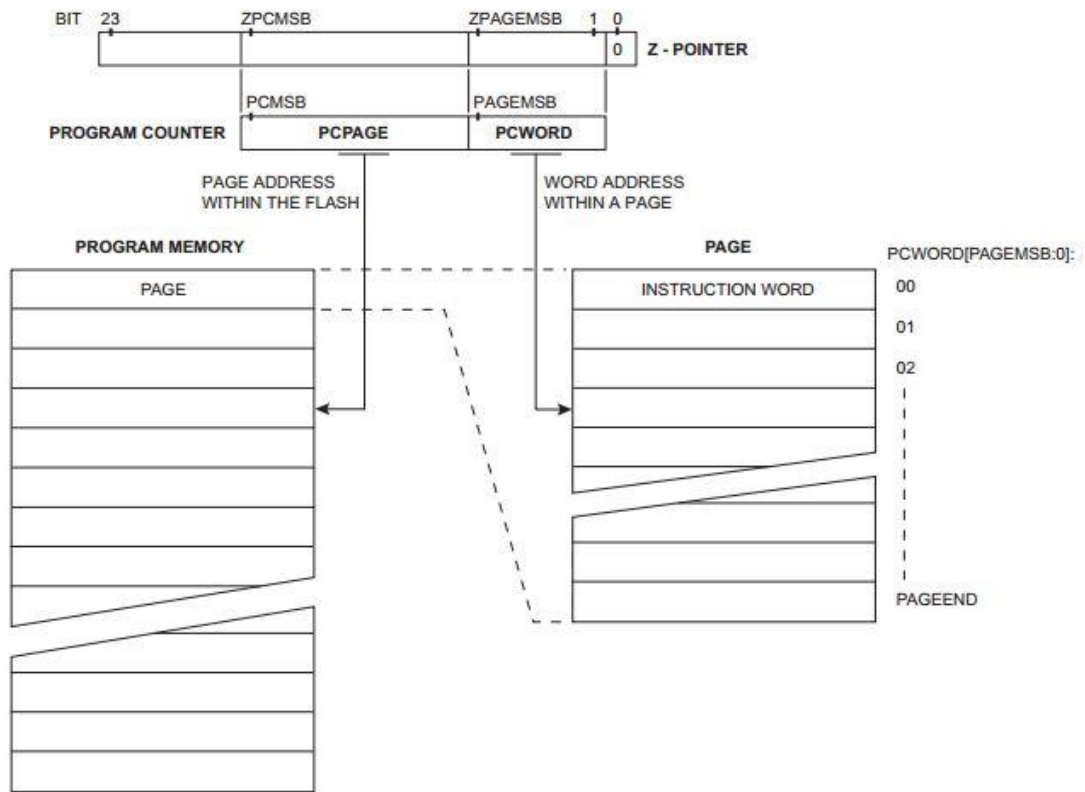


Figura 2.45. Direccionamiento de la memoria Flash durante SPM



# Capítulo 3:

# Desarrollo Software

## 3.1 Introducción.

En este capítulo se presenta una descripción a nivel funcional del *firmware* desarrollado en este Trabajo Fin de Grado. Así, en primer lugar se plantea una visión general del funcionamiento de la plataforma, describiéndose posteriormente el funcionamiento del sistema de interrupciones, así como la biblioteca CAN desarrollada. Por último se explican detalladamente las aplicaciones *router* y *bootloader* de los nodos, implementadas en este TFG.

## 3.2 Descripción general.

La plataforma experimental desarrollada se compone de tres elementos claramente diferenciables: el primer elemento es el Servidor central, el segundo elemento es el *router* que comunica dicho servidor con el bus CAN, y el último elemento estaría constituido por los nodos inalámbricos que se encuentran conectados al bus. En la Figura 3.1 se muestra un diagrama de bloques de la plataforma.

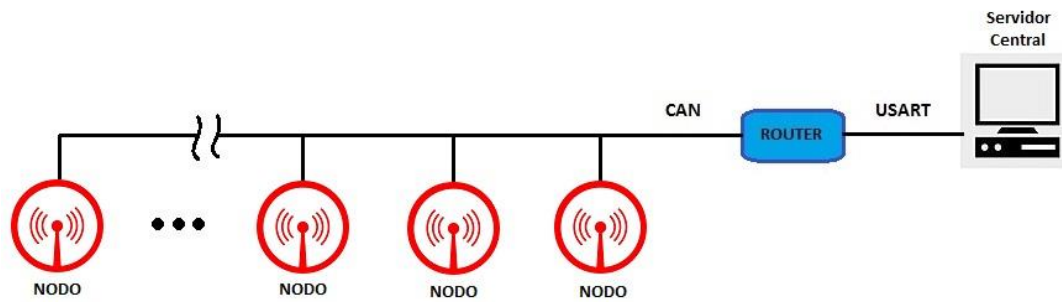


Figura 3.1. Estructura de la plataforma

A continuación se describe el proceso para programar uno de los nodos inalámbricos mediante la plataforma propuesta, que se encuentra representado en el diagrama de flujo de la Figura 3.2. Así, en primer lugar se toma el *firmware* que ha sido desarrollado (archivo *.hex* generado por el compilador) y se le pasa a la aplicación del Servidor, la cual lo analiza y divide en pequeños fragmentos de código. Esto se hace necesario debido a que el bus CAN sólo permite tramas con un tamaño máximo de 8 bytes, por lo que no es posible enviar todo el código en una sola transmisión. En segundo lugar se selecciona la dirección del nodo que se desea programar y se envía al *router* un comando que establece esa dirección. El tercer paso consiste en tomar los fragmentos de código, formar con ellos los distintos comandos que posteriormente deben interpretar los nodos, y enviarlos al *router*. En cuarto lugar el *router* recibe los comandos del servidor a través de la USART, los convierte en tramas CAN, y los envía por este bus a la dirección previamente establecida. En quinto lugar los nodos, una vez hayan recibido un comando, lo interpretan y llevan a cabo la tarea correspondiente. Por último, cualquiera de los nodos puede enviar mensajes al servidor a través del *router*; si este es el caso, el *router* toma la trama CAN recibida, la convierte en un mensaje serie, y lo transmite al Servidor a través de la USART.

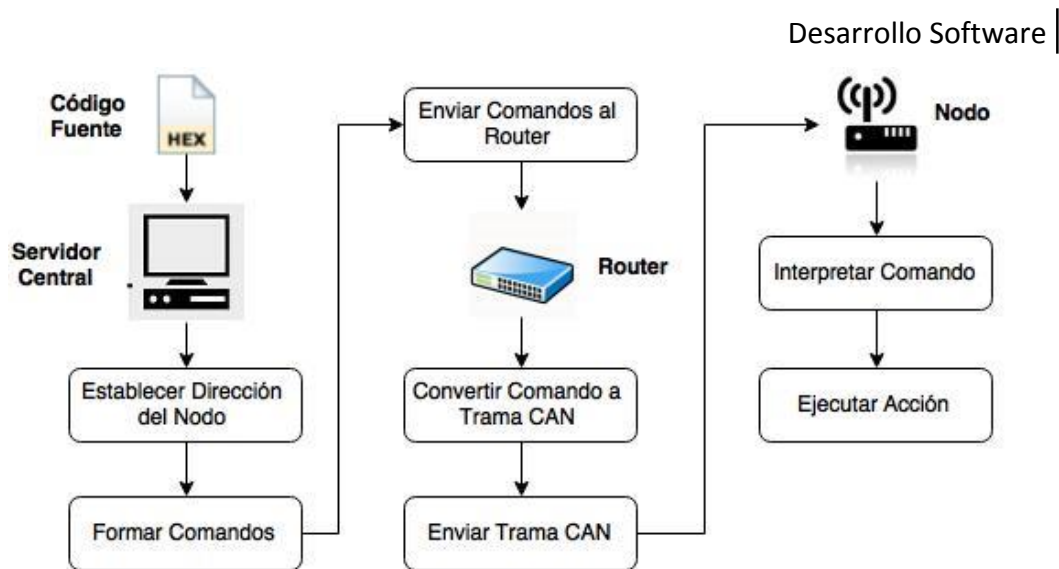


Figura 3.2. Diagrama de flujo de la programación de un nodo

La aplicación que se ejecuta en el Servidor, el cuál en principio es un ordenador común, ha sido desarrollada previamente por los profesores tutores y no forma parte de este TFG. Dicha aplicación ofrece una serie de opciones sobre las cuales se basa la implementación de las aplicaciones *router* y *bootloader* desarrolladas. En primer lugar permite la conexión con el *router* a través de la USART, estableciendo una comunicación directa entre ambos dispositivos a través de este bus serie. En segundo lugar es necesario seleccionar la dirección del nodo inalámbrico que se desea programar. Finalmente se dispone básicamente de tres opciones: programar el nodo, borrar la memoria *Flash* y ejecutar la aplicación remota.

De esta forma se dispondrá de un Servidor desde el cual será posible programar todos los nodos conectados al bus, haciendo uso de un *router* que permitirá direccionar cada uno de ellos, además de poder recibir mensajes provenientes de dichos nodos. Con esta finalidad se desarrollará el *firmware* correspondiente a la aplicación *router* y a la aplicación *bootloader* de los nodos inalámbricos. De esta forma se creará una infraestructura que permitirá programar y depurar los nodos de la plataforma de manera remota, concentrando los datos en un Servidor central.

### 3.3 Sistema de interrupciones.

Las aplicaciones *router* y *bootloader* desarrolladas tienen un comportamiento similar. Así, su funcionamiento normal está contenido en un bucle infinito en el cual realizan una serie de tareas periódicas. Por otro lado, tanto la USART como el CAN han sido configurados para trabajar mediante interrupciones. Así, la aplicación se encontrará normalmente realizando las tareas del bucle infinito, pudiendo ser interrumpidas en cualquier momento por uno de los buses, pasando entonces a atender la rutina de servicio correspondiente.

En los siguientes apartados se procederá a explicar en detalle cada una de las rutinas de servicio de las interrupciones asociadas a los buses, comenzando con las interrupciones relacionadas con el bus CAN, y continuando con las relacionadas con la USART.

#### 3.3.1 Vector de interrupción CAN.

En el microcontrolador AT90CAN128 existe un solo vector de interrupción asociado al bus CAN, tanto para la recepción como para la transmisión, motivo por el que se hace necesario atender ambos eventos dentro de la misma rutina de servicio. En la Figura 3.3 se puede observar un diagrama de flujo de la rutina de servicio CAN desarrollada. El código fuente de esta rutina puede encontrarse dentro de los ficheros *CAN128ROUTER.c* y *CAN128BOOTLOADER.c* de la aplicación *router* y la aplicación *bootloader*, respectivamente.

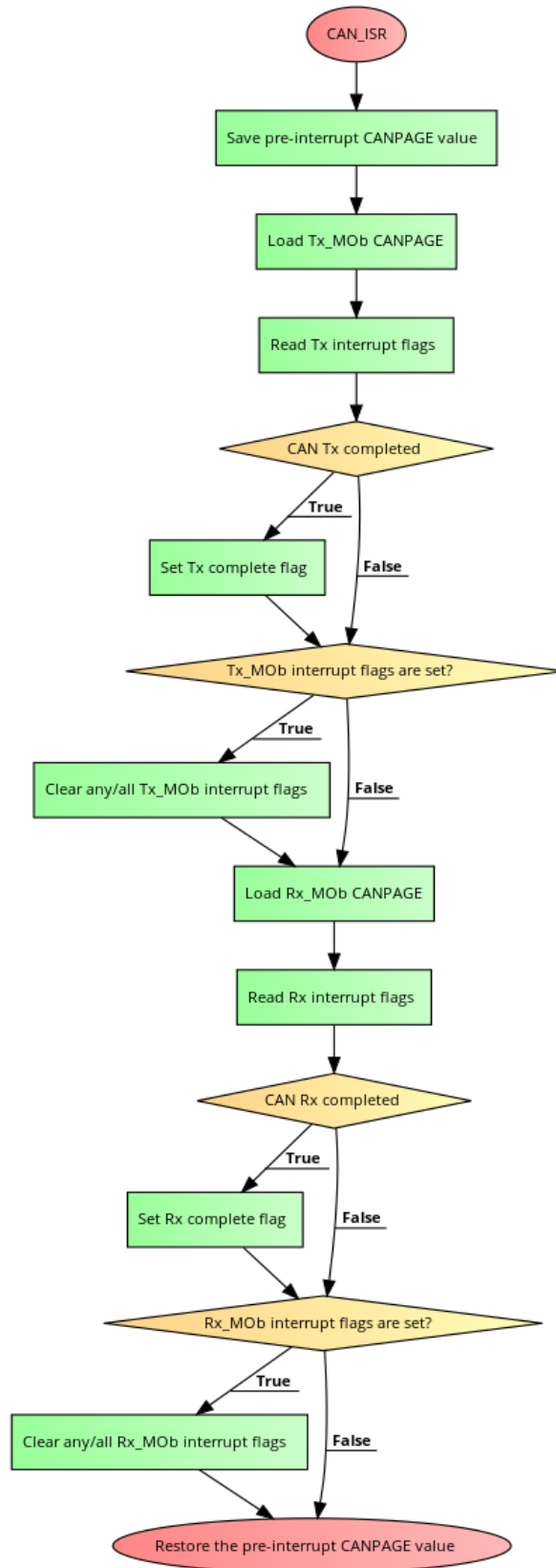


Figura 3.3. Diagrama de flujo de la rutina de servicio CAN

Como se puede observar, en primer lugar se almacena la página CAN que se encuentra seleccionada al entrar en la rutina de servicio. En segundo lugar se carga la página correspondiente a la transmisión y se guardan los *flags* de interrupción asociados a ella. En tercer lugar se comprueba si la interrupción ha sido generada por una transmisión completada; en caso afirmativo se activa el *flag* que indica una transmisión completada. Un cuarto paso sería limpiar los *flags* de interrupción, siempre que alguno de ellos se encuentre activado. A continuación se repite este mismo proceso pero para el caso de la recepción. Por último se restaura la página CAN que fue guardada al inicio, finalizando la rutina de servicio.

### 3.3.2 Vector de interrupción USART: transmisión.

En el caso de la USART, sí que se dispone de vectores de interrupción diferenciados para la transmisión y la recepción. Antes de pasar a describir la rutina de servicio correspondiente a la transmisión, se explicará el proceso para enviar un carácter a través de la USART.

Para empezar, se ha redireccionado la salida estándar, de forma tal que cada vez que se utiliza la función `printf()` los datos son enviados por la USART. Así, en primer lugar, se almacenan los caracteres a enviar en un buffer circular estructurado como una FIFO (*First In First Out*). Puesto que la USART ha sido configurada para enviar 8 bits, cada transmisión se corresponderá con un carácter. En segundo lugar se llama a la función que transmite un carácter, lo cual, una vez se ha completado la transmisión genera una interrupción. Por último, la rutina de servicio de la interrupción inicia una nueva transmisión en caso de que haya más caracteres en la FIFO. El código fuente de las funciones que realizan estas acciones puede encontrarse dentro de los ficheros `CAN128ROUTER.c` y `CAN128BOOTLOADER.c` de la aplicación *router* y la aplicación *bootloader*, respectivamente.



Para insertar los caracteres en la FIFO se utiliza la función `insertTxFIFO()`, de la cual se puede observar su diagrama de flujo en la Figura 3.4.

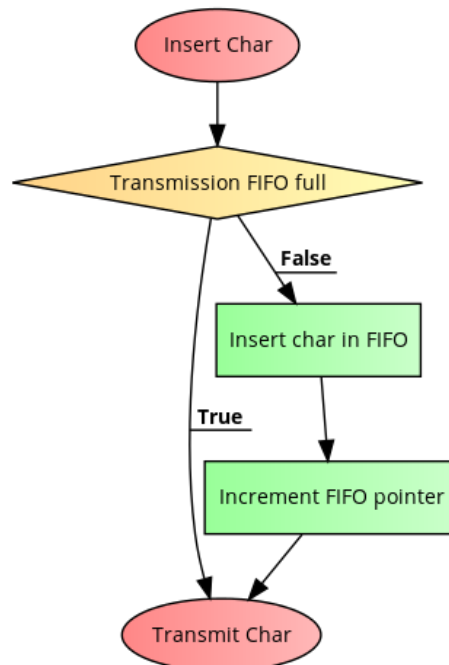


Figura 3.4. Diagrama de flujo de la función `insertTxFIFO`

En esta función se comprueba en primer lugar que la FIFO de transmisión no se encuentre llena, en cuyo caso no es posible agregar el elemento al buffer y simplemente se invoca la función de transmitir un carácter. En segundo lugar, en caso de que haya espacio en la FIFO, se almacena el carácter en el buffer. El tercer paso es incrementar el puntero utilizado para gestionar la FIFO circular. Por último, se invoca la función de transmitir un carácter.

Para transmitir un carácter almacenado en la FIFO se utiliza la función `TxChar()`, de la cual se puede observar su diagrama de flujo en la Figura 3.5.

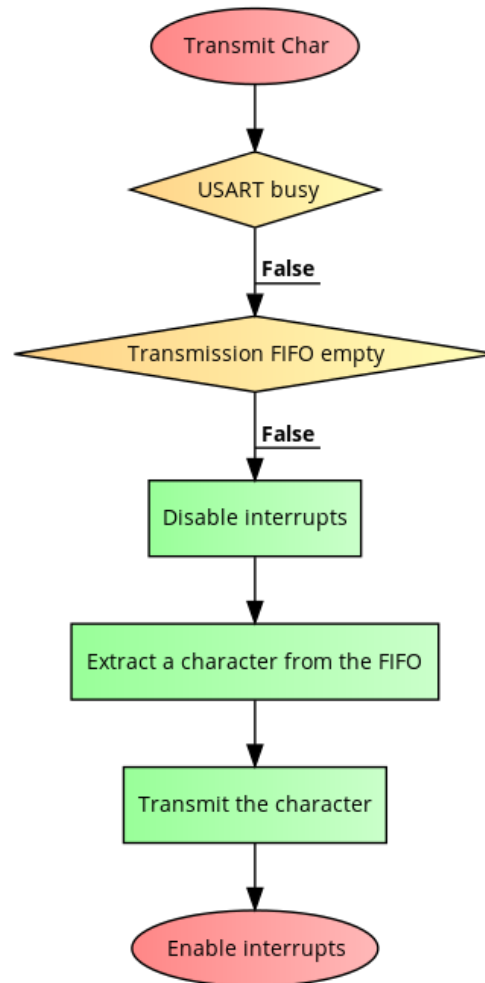


Figura 3.5. Diagrama de flujo de la función TxChar

En esta función, en primer lugar se comprueba si la USART se encuentra libre para realizar una transmisión. En segundo lugar se verifica si hay datos disponibles para transmitir en la FIFO. El tercer paso consiste en desactivar las interrupciones para evitar problemas de corrupción de datos. Un cuarto paso sería extraer un carácter de la FIFO. El siguiente paso sería colocar dicho carácter en el registro de transmisión de la USART, lo cual inicia la transmisión. Finalmente, se activan nuevamente las interrupciones.

Una vez visto esto, es posible pasar a explicar la rutina de servicio correspondiente a la transmisión de la USART, presentándose en la Figura 3.6 su diagrama de flujo. Esta es

una rutina de servicio en la que se invoca la función de transmitir un carácter, lográndose con ello transmitir continuamente hasta que no haya más elementos disponibles en la FIFO. El código fuente de esta rutina puede encontrarse dentro de los ficheros *CAN128ROUTER.c* y *CAN128BOOTLOADER.c* de la aplicación *router* y la aplicación *bootloader*, respectivamente.

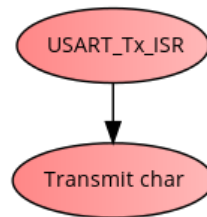


Figura 3.6. Diagrama de flujo de la rutina de servicio USART: transmisión

### 3.3.3 Vector de interrupción USART: recepción.

Finalmente el otro vector de interrupción de la USART corresponde a la recepción. Esta interrupción solo es utilizada en el *router*, ya que los nodos inalámbricos no reciben información a través del bus serie. En la rutina de servicio se extraerá el carácter recibido y se comprueba si pertenece a un grupo de caracteres determinado, para llevar a cabo las acciones correspondientes. En la Figura 3.7 se presenta un diagrama de flujo en el que se representa gráficamente cómo ha sido implementado este proceso. El código fuente de esta rutina puede encontrarse dentro del fichero *CAN128ROUTER.c* de la aplicación *router*.

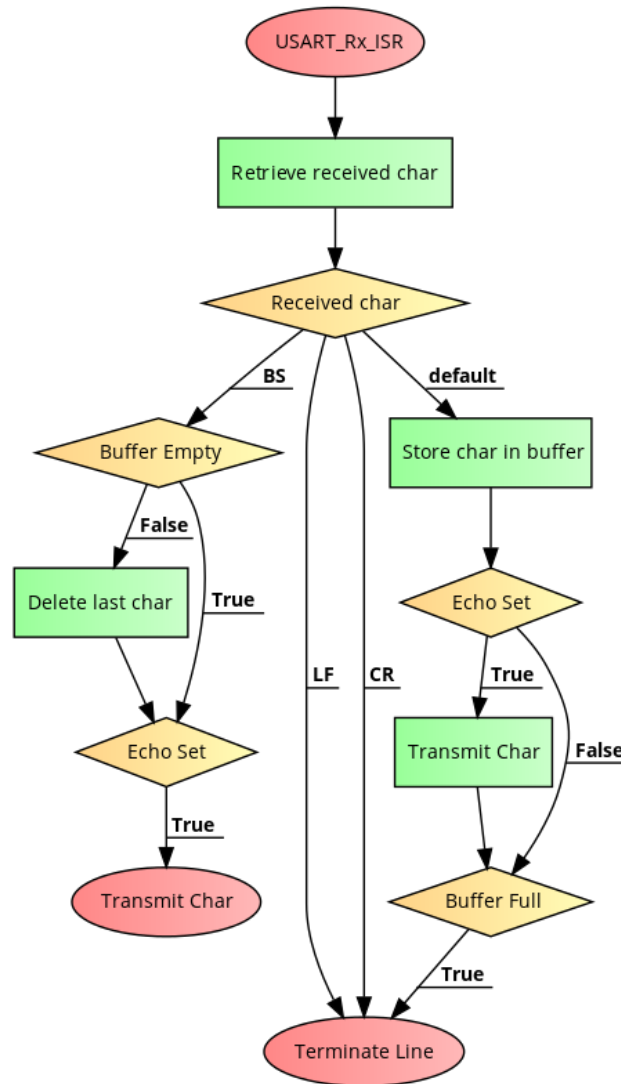


Figura 3.7. Diagrama de flujo de la rutina de servicio USART: recepción

En este caso, en primer lugar se almacena el carácter recibido en una variable local. En segundo lugar se comprueba a qué grupo de caracteres pertenece, con el fin de realizar las acciones correspondientes. El primero de estos grupos es el carácter de borrar (*BackSpace*, BS), en cuyo caso se disminuye en uno la variable que indica el número de elementos disponibles en el buffer de recepción, con lo cual a efectos prácticos se borra el último carácter que se haya recibido; si el buffer está vacío, no se hace nada, además si está activado el eco, se transmite este mismo carácter de vuelta a la USART. El segundo grupo de caracteres es el conjunto formado por *Carry Return* (CR) y *Line Feed* (LF). En caso

de que el carácter recibido sea alguno de estos dos, entonces se invoca la función de terminar la línea recibida, `terminateRx()`, la cual se explicará más adelante. El último grupo lo componen el resto de caracteres que no sean los descritos anteriormente, en cuyo caso se almacena el carácter en el buffer de recepción, posteriormente si está activado el eco se transmite este mismo carácter de vuelta a la USART, y finalmente se comprueba si se ha llenado el buffer, en cuyo caso se invoca a la función de terminar la línea recibida, `terminateRx()`. En la Figura 3.8 se puede observar el diagrama de flujo de la función `terminateRx()`.

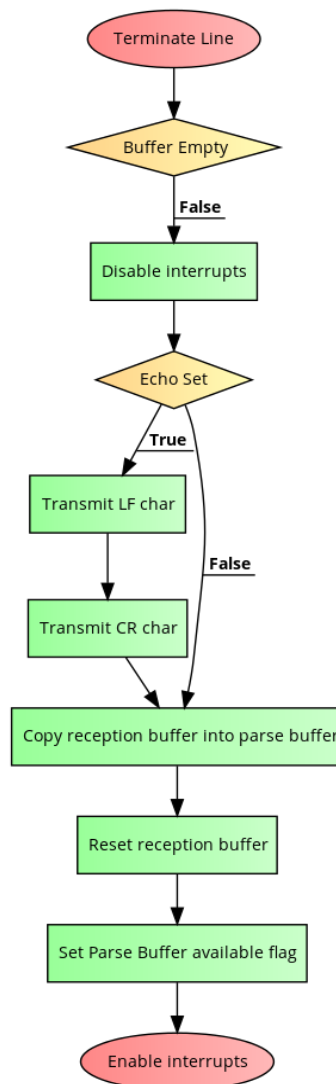


Figura 3.8. Diagrama de flujo de la función `terminateRx`

Esta es la función que se invoca una vez se desea procesar la cadena de caracteres que se ha recibido, la cual se encuentra almacenada en el buffer de recepción. En primer lugar se comprueba que el buffer no esté vacío, en cuyo caso simplemente se sale de la función sin hacer nada. En segundo lugar se desactivan las interrupciones para evitar la corrupción de los datos, puesto que podría darse el caso de que se reciban más caracteres mientras se está dentro de la función, modificando de esta forma el contenido original que se quería procesar. En tercer lugar se comprueba si está activado el eco, en cuyo caso se transmiten los caracteres LF y CR de vuelta a la USART. El cuarto paso consiste en copiar el contenido del buffer de recepción en un buffer auxiliar, que será el utilizado en el programa principal para interpretar los comandos. A continuación se indica que el buffer de recepción está vacío nuevamente y se activa el *flag* que indica que hay un comando listo para procesar. Por último se vuelven a activar las interrupciones y se termina la función.

De esta forma se cuenta, por un lado, con un comando listo para ser procesado, el cual se almacena en un buffer auxiliar, mientras que por el otro lado, el buffer de recepción vuelve a estar disponible para recibir una nueva línea de comandos. Con ello se logra poder recibir un nuevo comando mientras el programa principal procesa el que ha sido recibido previamente.

### 3.4 Biblioteca CAN.

Como parte del presente TFG se ha desarrollado una biblioteca que facilite el uso del bus CAN, para lo cual se ponen a disposición de los usuarios una serie de funciones públicas, definidas en el archivo de cabeceras, que controlan el bus de forma transparente. El código fuente de la biblioteca CAN desarrollada se encuentra dentro de los archivos *CAN128LIB.c* y *CAN128LIB.h*.

### 3.4.1 Configuración previa.

Antes de presentar las funciones de la biblioteca CAN disponibles para el usuario, hay una serie de aspectos previos a tener en cuenta. En el archivo de cabeceras ("`can_lib.h`") hay dos definiciones que se deben adaptar al caso concreto de la aplicación que se vaya a desarrollar. La primera de ellas es la frecuencia del oscilador, debido a que los módulos sensores inalámbricos utilizados en este TFG cuentan con un oscilador a 16MHz, en la biblioteca se ha establecido por defecto esa frecuencia de la siguiente forma:

```
#define FOSC 16000000UL
```

El otro elemento que se debe definir antes de usar la biblioteca CAN desarrollada es la tasa de baudios del bus. En el caso de este TFG se ha trabajado con una tasa de 100Kbps, definida de la siguiente forma:

```
#define CAN_BAUDRATE 100
```

Actualmente la biblioteca soporta frecuencias de oscilador de 16MHz, 12MHz y 8MHz, además de velocidades de bus de 100Kbps, 125Kbps, 200Kbps, 250Kbps, 500Kbps y 1Mbps para cada una de estas frecuencias de oscilador.

### 3.4.2 Funciones públicas: configuración del bus.

Una vez hayan sido configurados estos elementos, se puede pasar a hacer uso de la biblioteca CAN. En ella es posible diferenciar dos partes, las funciones dedicadas a la transmisión y las dedicadas a la recepción, siendo ambas completamente independientes entre sí. De esta forma, en caso de tener que utilizar únicamente una de ellas, se puede eliminar la otra, pudiendo ahorrar memoria en caso necesario.

Antes de poder enviar y recibir por el bus CAN se hace necesario configurarlo e inicializarlo, para lo cual se proporciona una de las funciones públicas. Esta función, con cabecera `void initCAN()` se encarga de configurar y habilitar el controlador CAN integrado en el microcontrolador, haciendo uso de los parámetros definidos previamente, además de inicializar todas las páginas CAN a sus valores por defecto. Un elemento a tener en cuenta es que esta configuración establece el uso de interrupciones como forma de gestionar los eventos, de forma que los usuarios necesitarán definir en sus aplicaciones la rutina de servicio correspondiente.

### 3.4.3 Funciones públicas: transmisión.

Una vez realizada la configuración del bus es posible comenzar, tanto la transmisión como la recepción. En el caso de la transmisión, se cuenta con tres funciones que permitirán establecer la dirección CAN, añadir un elemento al buffer de transmisión, y por último, iniciar una transmisión.

La primera de estas funciones es la de establecer la dirección CAN, la cual tiene la siguiente cabecera: `void set_can_address(unsigned char Can_base_id)`. Esta función requiere de un parámetro que es la dirección del nodo que va a transmitir la trama. A pesar de que la implementación se ha realizado estableciendo direcciones CAN de 29 bits, el usuario no se tiene que preocupar por esto, teniendo que proporcionar la dirección como un número entero, siendo la función la encargada de confeccionar el identificador adecuado. El identificador se conforma con un formato origen-destino, de forma que la mitad superior de los 29 bits del identificador corresponden a la dirección del nodo que envía la trama, mientras que la mitad inferior corresponde al destino de la trama. Como la plataforma se ha diseñado para que los nodos sólo puedan enviar tramas hacia el *router*, los 14 primeros bits correspondientes al destino siempre serán 0, que es la dirección asignada por defecto al *router*.



La siguiente función es la de añadir un carácter en el buffer de transmisión, que tiene la siguiente cabecera: `int can_putchar(char c)`. Esta función toma el carácter que se le pasa por parámetro y lo inserta en el buffer de transmisión, que está implementado como una FIFO circular. Además de esto, una vez se añade el elemento a la FIFO, también se inicia la transmisión de una trama.

Como elemento adicional se ha definido esta misma función con el formato necesario en caso de que se deseara redirigir la salida estándar hacia el CAN: `static int can_putchar(char c, FILE *stream)`. Esta función en principio se encuentra comentada, quedando a disposición del usuario para su uso.

Por último se tiene la función que inicia la transmisión de una trama CAN, cuya cabecera es la siguiente: `void CanTx()`. En este caso lo que se hace es tomar los caracteres que haya disponibles en la FIFO, hasta un máximo de 8 que permite la trama CAN, y se transmite el identificador que fue establecido previamente; en caso de que la FIFO esté vacía, no se hace nada y se termina la función.

### 3.4.4 Funciones públicas: recepción.

Las funciones destinadas a la recepción también son tres, comenzar la recepción, guardar la trama recibida en el buffer de recepción, y por último, extraer un carácter del buffer.

La primera de estas funciones es la de iniciar la recepción, la cual tiene la siguiente cabecera: `void can_rx_mob(uint32_t Can_base_id)`. Esta función requiere de un parámetro que es la dirección del nodo. Al igual que en el caso de la transmisión, la dirección es de 29 bits y tiene un formato origen-destino, siendo los últimos 15 bits la dirección de quien envía la trama y los primeros 14 bits la dirección del destino. Puesto que

en nuestra plataforma experimental los nodos solo pueden recibir tramas provenientes de la pasarela, la dirección origen en este caso siempre será 0. Una vez se invoca esta función, el nodo comenzará a aceptar tramas y comparar las direcciones con la que tiene programada, generándose una interrupción en caso de coincidencia.

La siguiente función es la de guardar la trama recibida en el buffer de recepción, siendo su cabecera la siguiente: `void can_retrieve_frame()`. En este caso no hay ningún parámetro, y simplemente se debe invocar la función cada vez que se reciba una trama, con lo cual se extraen los datos recibidos y se almacenan en el buffer. Tener en cuenta que si en un momento determinado el buffer se llena, los datos que se reciban a continuación se perderán, ya que el control del estado del buffer queda en manos del usuario.

Por último se tiene la función que extrae un carácter del buffer, cuya cabecera es la siguiente: `char can_getchar()`. Esta función no tiene parámetros y devuelve el primer elemento del buffer, es decir, el carácter más antiguo que se haya recibido.

### 3.5 Aplicación router.

Como se indicó al inicio del capítulo, el módulo *router* es el encargado de recibir los comandos provenientes del Servidor a través de la USART y enviarlos al nodo destino que se encuentra conectado al bus CAN. Por otra parte, también hace el proceso inverso, es decir, toma los mensajes que envían los nodos y se los pasa al Servidor. En la Figura 3.9 se puede observar el diagrama de flujo que sigue el programa principal, la función `int main(void)`, durante la cual se configura el microcontrolador y sus periféricos, además de realizarse todas las inicializaciones necesarias, para luego entrar en el bucle infinito. El código fuente de la aplicación *router* puede encontrarse dentro de los ficheros *CAN128ROUTER.c* y *CAN128ROUTER.h*.

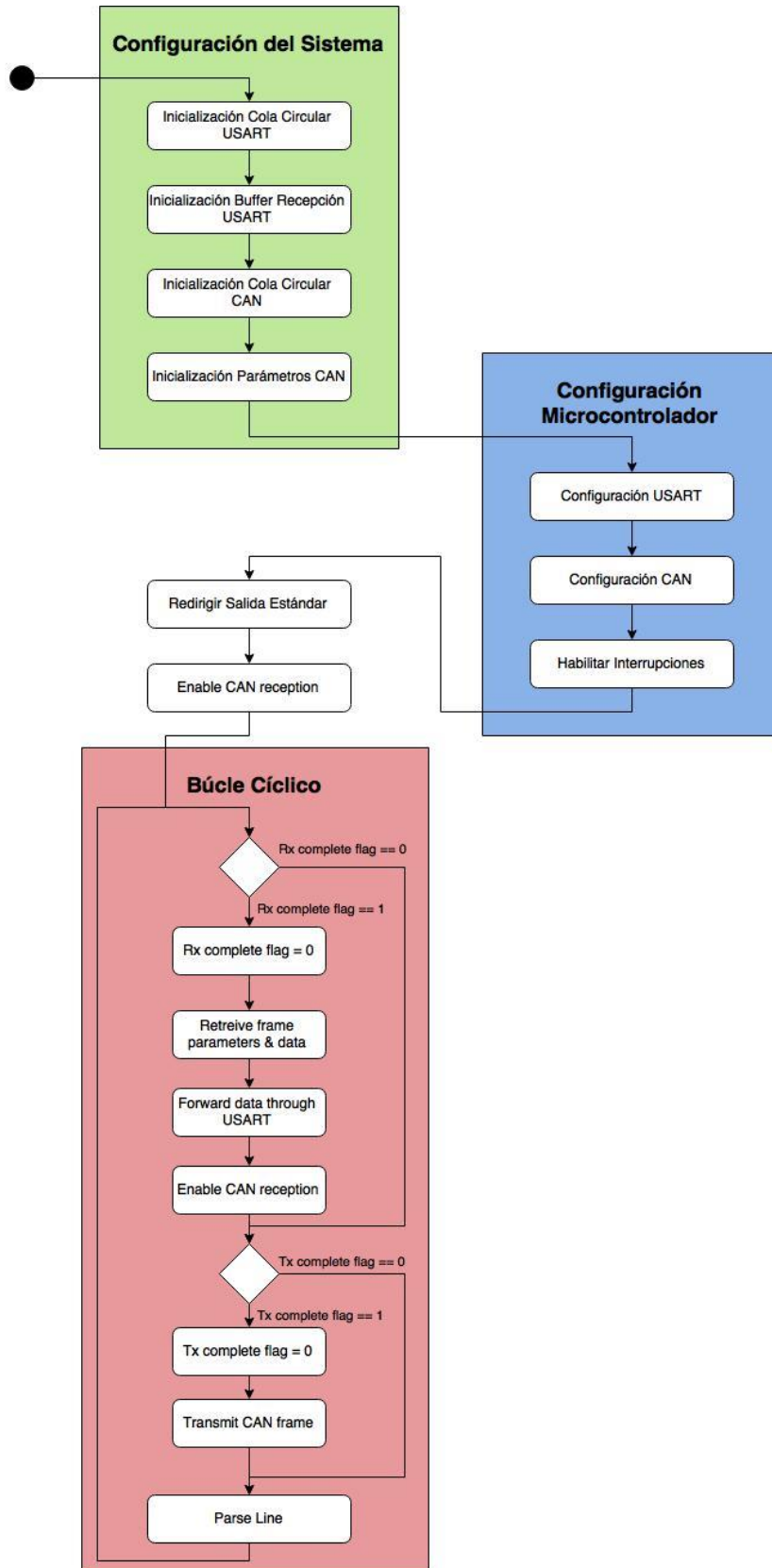


Figura 3.9. Diagrama de flujo de la aplicación router

### **Configuración del Sistema.**

La configuración del sistema consiste en una serie de inicializaciones de los distintos elementos que serán utilizados para gestionar los periféricos. En primer lugar, se inicializa la cola circular en la cual se almacenarán los datos para ser enviados a través de la USART. En segundo lugar se inicializa el buffer donde se guardarán todos los datos que se reciban a través de la USART. El tercer paso es inicializar la cola circular en la que se almacenarán los datos para ser enviados a través del bus CAN. El último paso consiste en inicializar una serie de parámetros relacionados con el controlador CAN, como pueden ser la dirección base, la máscara de recepción, etc.

### **Configuración del Microcontrolador.**

En este bloque se realiza la inicialización hardware. En primer lugar se lleva a cabo la configuración de la USART, que será el medio de comunicación con el servidor. La configuración de este puerto es básica, ya que en ella se seleccionan los parámetros de la comunicación serie, tales como la tasa de transferencia de datos, la longitud de los datos, la paridad, bits de *stop*, y el modo de funcionamiento. Para este TFG se ha definido una velocidad de 38400 baudios, 8 bits de datos, sin paridad, 1 bit de *stop* y que sea gestionado mediante interrupciones. Estos parámetros fueron elegidos puesto que, cuando se llevaron a cabo las pruebas iniciales, se comprobó que la USART funcionaba correctamente con el resto de la plataforma, y por tanto se mantuvo esta configuración.

En segundo lugar se procede a la configuración del controlador CAN, que será el medio de comunicación con los nodos inalámbricos. La configuración de este controlador abarca aspectos tales como la velocidad del bus, inicialización de las páginas CAN y modo de funcionamiento. Además, como parte de la configuración del controlador hay que configurar el pin GPIO que actúa como *enable* del transceptor CAN externo. Para este TFG se ha establecido una velocidad de bus de 100Kbps, una página dedicada a la recepción y otra para la transmisión, igualmente gestionado mediante interrupciones. Finalmente, el

último paso consiste en habilitar las interrupciones generales. En este caso, se estableció la velocidad más baja del bus para evitar posibles cuellos de botella en el *router*, además de dedicar una página CAN a la transmisión y otra a la recepción para facilitar la gestión del controlador CAN.

A continuación, y como paso previo antes de entrar al bucle cíclico, se redirige en primer lugar la salida estándar hacia la USART, de forma que cada vez que se invoque la función `printf()` los datos sean enviados a través de la USART, y en segundo lugar se inicia la recepción CAN, con lo cual se comienzan a analizar todas las tramas del bus, y en caso de que estén dirigidas al *router*, se genera una interrupción.

### **Bucle Cíclico.**

El siguiente bloque dentro de la función principal corresponde con el bucle infinito. En apartados anteriores se ha explicado que cuando una interrupción es generada en el bus CAN, se activa un *flag* para indicar que se ha completado la recepción o transmisión, siendo en este bucle donde se atienden estos.

En este bloque se comprueba en primer lugar el estado del *flag* que indica que se ha completado una recepción CAN. En caso de que se encuentre activo, se llevan a cabo las siguientes acciones. En primer lugar, se desactiva el *flag* para indicar que la recepción ha sido atendida. Lo segundo es extraer los parámetros asociados a la trama, que son la dirección y número de bytes de datos (DLC) que contiene la misma. En tercer lugar se extraen los bytes de datos y se almacenan en un buffer temporal. En cuarto lugar se realiza un eco a través de la USART de la trama recibida, añadiendo una cabecera con la dirección del nodo que envía el. Por último se vuelve a activar la recepción en el controlador CAN.

En segundo lugar se comprueba el estado del *flag* que indica que se ha completado una transmisión CAN. En caso de que se encuentre activo, lo primero es desactivar el *flag* para indicar que ya ha sido atendida dicha transmisión. A continuación, se inicia una nueva transmisión, lográndose de esta forma estar transmitiendo continuamente hasta que no haya más elementos disponibles en la FIFO. Dentro de la función de transmisión se comprueba si la FIFO está vacía o no; una vez se haya vaciado, se da por terminada la transmisión.

El último elemento dentro del bucle infinito es una llamada a la función *Parse Line*, encargada de tomar la línea de comandos enviada por el Servidor, e interpretarla para llevar a cabo las acciones correspondientes. En la Figura 3.10 se observa un diagrama de su flujo de ejecución.

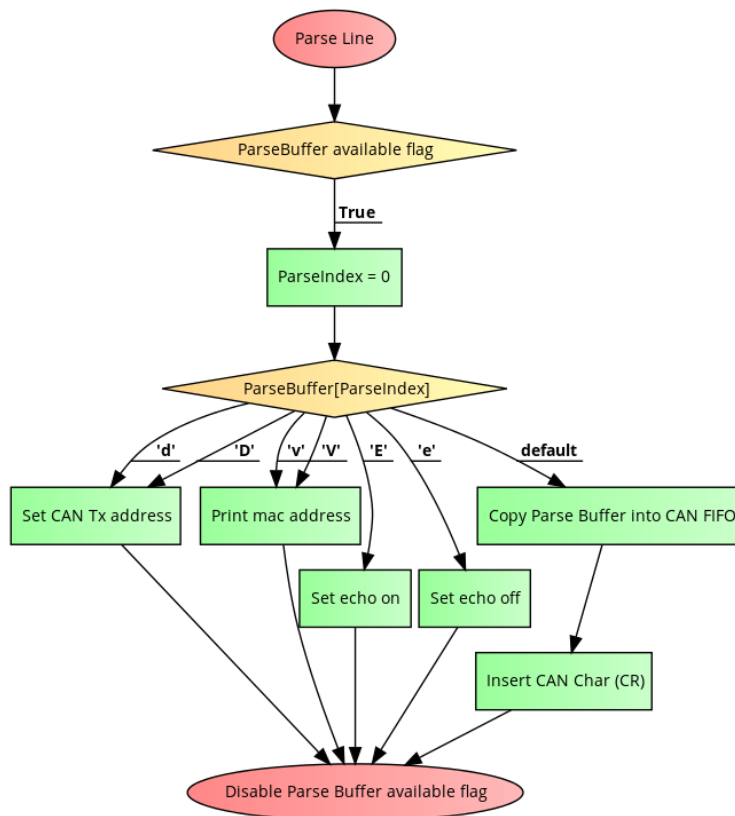


Figura 3.10. Diagrama de flujo de la función *parseLine*

En primer lugar se comprueba el estado del *flag* que indica que hay una línea lista para procesar. En caso de que esté activo, en segundo lugar lo que se hace es reiniciar el índice que recorre el buffer de recepción. Al tratarse del *router*, los comandos pueden estar dirigidos, bien al propio *router*, o a alguno de los nodos. Es por esto que el tercer paso consiste en identificar el comando, que puede ser: fijar la dirección CAN del nodo que se desea programar, imprimir la dirección MAC del *router*, activar o desactivar el eco por la USART, o enviar la línea de comandos hacia el nodo inalámbrico. Por último se desactiva el *flag* para indicar que la línea ya ha sido procesada, y se sale de la función.

### 3.6 Aplicación bootloader.

Los nodos inalámbricos son los dispositivos finales que estarán conectados al bus CAN, los cuales están constantemente a la escucha del bus esperando recibir los comandos provenientes del servidor. El *firmware* de los nodos está alojado en la zona de *bootloader*, dejando el resto de la memoria *Flash* disponible para almacenar la aplicación de usuario. En la Figura 3.11 se puede observar el diagrama de flujo que sigue el programa principal, la función `int main(void)`, en la que se configura el microcontrolador y sus periféricos, además de realizarse todas las inicializaciones necesarias, para luego entrar en el bucle infinito. El código fuente de la aplicación *bootloader* puede encontrarse dentro de los ficheros *CAN128BOOTLOADER.c* y *CAN128BOOTLOADER.h*.

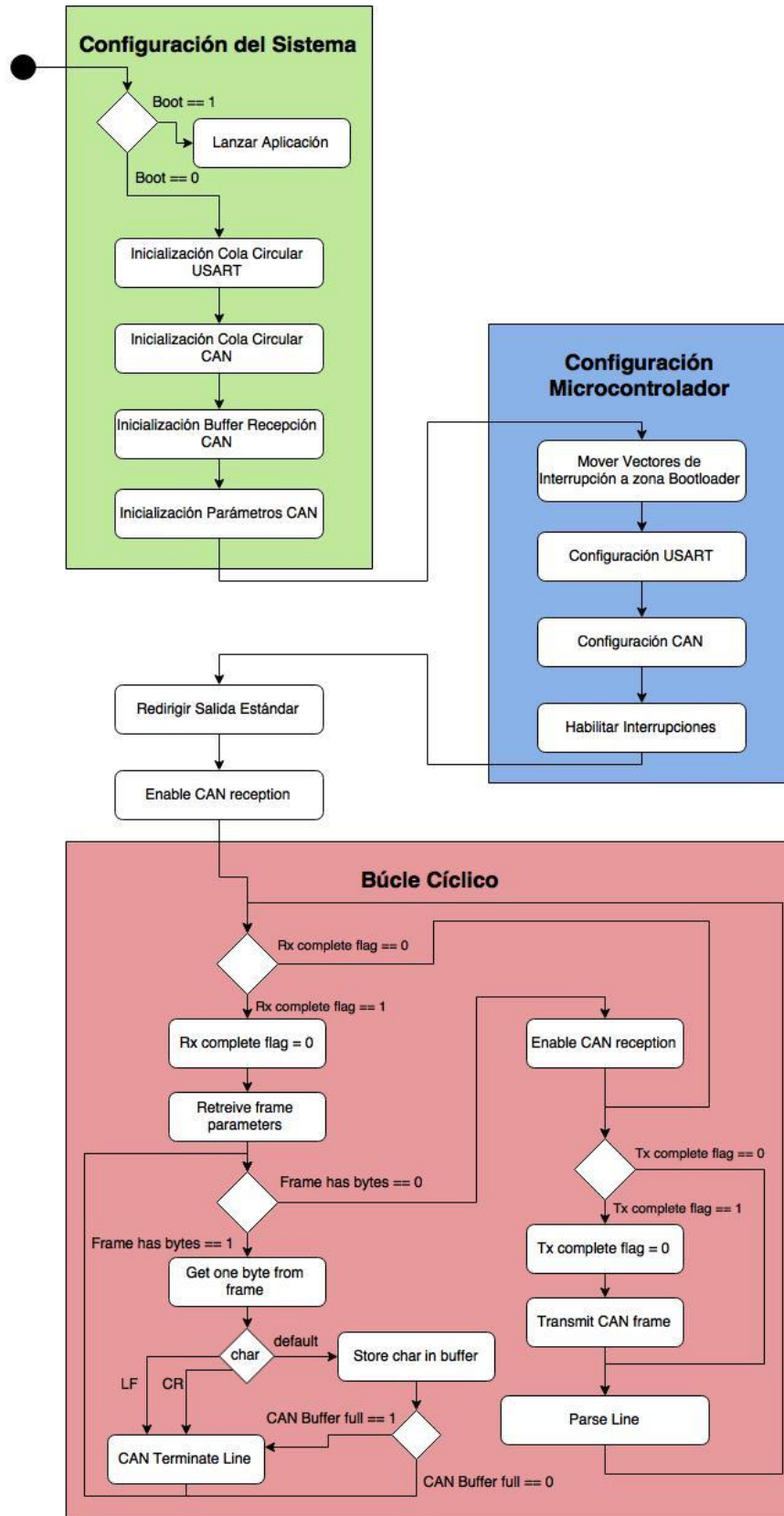


Figura 3.11. Diagrama de flujo de la aplicación bootloader



### **Configuración del Sistema.**

La configuración del sistema consiste en una serie de inicializaciones de los distintos elementos que serán utilizados para gestionar los periféricos. En primer lugar, se comprueba el pin GPIO que indica si se debe lanzar la aplicación o entrar en modo *bootloader*. En caso de que se lance dicha aplicación, los siguientes pasos representados en el diagrama no se llevarán a cabo. En segundo lugar se inicializa la cola circular en la que se almacenarán los datos para ser enviados a través de la USART. El tercer paso consiste en inicializar la cola circular en la cual se almacenarán los datos para ser enviados a través del bus CAN. En cuarto lugar se inicializa el buffer donde se guardarán todos los datos que se reciban a través del bus CAN. El último paso consiste en inicializar una serie de parámetros relacionados con el controlador CAN, como pueden ser la dirección base, la máscara de recepción, etc.

### **Configuración del Microcontrolador.**

En este bloque se realiza la inicialización hardware. En primer lugar se mueven los vectores de interrupción a la zona de *bootloader*. Esto es necesario debido a que cuando se realiza alguna operación desde esta zona, no es posible acceder a ningún código que se encuentre localizado en la zona de la memoria destinada a la aplicación, que es donde se encuentran ubicados los vectores de interrupción por defecto.

En segundo lugar se realiza la configuración de la USART, que en el caso de los nodos ha sido utilizada como medio de depuración. La configuración de este puerto es básica, seleccionándose en ella los parámetros de la comunicación serie, como son la tasa de transferencia de datos, la longitud de los datos, la paridad, bits de *stop* y el modo de funcionamiento. Para este TFG se ha definido una velocidad de 38400 baudios, 8 bits de datos, sin paridad, 1 bit de *stop* y que sea manejado mediante interrupciones. Esta configuración fue elegida, puesto que permitía llevar a cabo correctamente las tareas de depuración.

En tercer lugar se procede a la configuración del controlador CAN, que será el medio de comunicación con el *router*. La configuración de este controlador abarca aspectos tales como la velocidad del bus, inicialización de las páginas CAN y modo de funcionamiento. Además, como parte de la configuración del controlador, hay que configurar el pin GPIO que actúa como *enable* del transceptor CAN externo. Para este TFG se ha establecido una velocidad de bus de 100Kbps, una página dedicada a la recepción y otra para la transmisión, igualmente gestionado mediante interrupciones. Finalmente, el último paso consiste en habilitar las interrupciones generales.

A continuación, y como pasos previos antes de entrar al bucle cíclico, se redirige en primer lugar la salida estándar hacia la USART, de manera que cada vez que se invoque la función `printf()` los datos sean enviados a través de la USART. En segundo lugar se inicia la recepción CAN, con lo cual se comienzan a analizar todas las tramas del bus, y en caso de que estén dirigidas al nodo en cuestión, se genera una interrupción.

### **Bucle Cíclico.**

El siguiente bloque dentro de la función principal corresponde con el bucle infinito. En apartados anteriores se ha explicado que cuando se genera una interrupción en el bus CAN, se activa un *flag* para indicar que se ha completado la recepción o transmisión, siendo en este bucle donde se atienden.

En este bloque se comprueba en primer lugar el estado del *flag* que indica que se ha completado una recepción CAN. En caso de que se encuentre activo, se llevan a cabo las siguientes acciones. Lo primero se desactiva el *flag* para indicar que la recepción ha sido atendida. Lo segundo es extraer los parámetros asociados a la trama, que son la dirección y número de bytes de datos (DLC) que contiene la misma. En tercer lugar se comprueba si aún quedan caracteres por extraer de la trama. En caso afirmativo, el cuarto paso consiste en extraer uno de estos caracteres. El quinto paso es comprobar si el carácter extraído se

corresponde con *Carry Return* (CR) o *Line Feed* (LF) lo cual indica el fin de una línea de comandos y que se debe procesar; en caso contrario se guarda el carácter en el buffer de recepción. Por último, una vez se han extraído todos los caracteres de la trama, se vuelve a activar la recepción en el controlador CAN.

En segundo lugar se comprueba el estado del *flag* que indica que se ha completado una transmisión CAN. En caso de que se encuentre activo, lo primero es desactivar el *flag* para indicar que ya ha sido atendida dicha transmisión. A continuación, se inicia una nueva transmisión, logrando así estar transmitiendo continuamente hasta que no haya más elementos disponibles en la FIFO. Dentro de la función de transmitir se comprueba si la FIFO está vacía o no, de forma que una vez se haya vaciado, se da por terminada la transmisión.

El último elemento dentro del bucle infinito es una llamada a la función *Parse Line*, encargada de tomar la línea de comandos enviada por el Servidor e interpretarla para llevar a cabo las acciones correspondientes. En la Figura 3.12 se observa un diagrama de su flujo de ejecución.

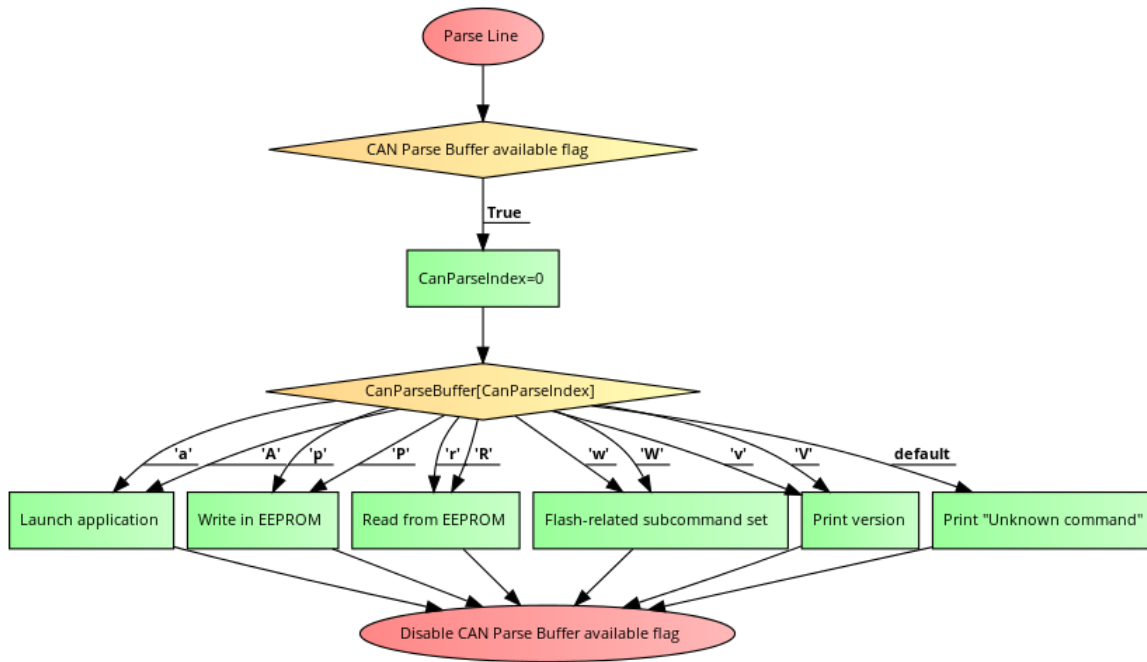


Figura 3.12. Diagrama de flujo de la función parseLine

En primer lugar se comprueba el estado del *flag* que indica que hay una línea lista para procesar. En caso de que esté activo, en segundo lugar lo que se hace es reiniciar el índice que recorre el buffer de recepción. El tercer paso consiste en identificar el comando, para lo cual se analiza el primer carácter de la línea; estos comandos pueden ser los siguientes: lanzar la aplicación de usuario, escribir en la EEPROM, leer de la EEPROM, el conjunto de comandos relacionados con la programación de la memoria *Flash*, imprimir la versión del *firmware*, o imprimir la cadena "Comando desconocido" cuando no se reconoce el comando. Por último se desactiva el *flag* para indicar que ya la línea ha sido procesada y se sale de la función.

El primero de los comandos es el de invocar la aplicación de usuario. En esta función se restablecen a los valores por defecto todos los registros internos del microprocesador que han sido utilizados por el *bootloader*, que son los relacionados con la USART y el CAN. Además, se mueven los vectores de interrupción de vuelta hacia el inicio de la memoria

*Flash*, y por último se salta a la dirección 0 de la memoria, con lo cual comienza la ejecución de la aplicación que fue programada previamente.

El segundo comando es el de escribir en la EEPROM. Este comando está compuesto por la dirección en la que se desea escribir y el byte que se desea almacenar. La función destinada a ello se encarga de comprobar que tanto la dirección como el dato a escribir sean válidos, ya que estos deben estar en formato hexadecimal.

El tercer comando es el de leer de la EEPROM. En este comando sólo es necesario indicar la dirección de la cual se desea leer, al igual que en el caso anterior, la función se encarga de comprobar que sea una dirección válida la que se ha indicado, además de devolver el byte que se encuentra en esa dirección. Destacar que estas dos funciones se han incluido como un extra, ya que este TFG se centra en programar la memoria *Flash*.

El cuarto es el subconjunto de comandos relacionados con la programación de la memoria *Flash*. Aclarar que para implementar estos comandos que gestionan la memoria se ha hecho uso de una biblioteca denominada "*sp\_driver*", proporcionada por el propio fabricante Atmel. Dicha biblioteca pone a nuestra disposición funciones para leer, borrar y escribir en la memoria *Flash*. Este subconjunto lo componen cinco comandos. El primero de ellos es el de cargar la dirección de la página que se va a programar. El segundo carga un dato dentro del buffer temporal. El tercero escribe el contenido de este buffer temporal en la página. El cuarto lo que hace es borrar la página seleccionada con el primer comando, mientras que el último comando permite leer un byte de una dirección de memoria determinada que se le pasa como parámetro.

El quinto comando es el de imprimir la versión del *firmware*. Este comando simplemente envía por la USART una cadena indicando la versión actual del *firmware*. Se

ha utilizado durante la fase de desarrollo como medio para aclarar visualmente qué versión del *firmware* es la que se está ejecutando.

Por último está el comando que imprime por la USART la cadena "**Comando desconocido**". Éste se utiliza cuando no se ha reconocido el comando recibido como ninguno de los explicados previamente. Al igual que el comando anterior, se ha implementado para facilitar el proceso de desarrollo y depuración.

# Capítulo 4:

# Resultados

Con el objetivo de validar el funcionamiento de las aplicaciones *router* y *bootloader* implementadas en el presente Trabajo Fin de Grado, se desarrollaron dos aplicaciones de ejemplo con las que se programaron los nodos inalámbricos haciendo uso de la plataforma experimental propuesta. Inicialmente se intentó programar un único nodo inalámbrico. Una vez fue posible hacer esto correctamente, se procedió a incluir un mayor número de nodos en la red. En la Figura 4.1 se puede observar el montaje práctico realizado para llevar a cabo estas pruebas.



Figura 4.1. Montaje práctico de la plataforma experimental

## | Resultados

Para el desarrollo del ejemplo aquí presentado, se han utilizado tres módulos sensores inalámbricos, de los cuales dos actúan como nodos inalámbricos conectados al bus CAN, y el restante actúa como *router*, que por un lado se comunica con el bus CAN y por el otro, con el Servidor central a través de la USART. Para cumplir con el estándar CAN, al nodo intermedio se le ha retirado la terminación resistiva del bus. Por otro lado, el Servidor central se ejecuta en un ordenador portátil.

La primera de estas pruebas consiste en realizar la programación de un único nodo, para lo cual se desarrolló una aplicación de ejemplo que imprime por la USART la cadena "Ho1a" al inicio de la función principal. Se decidió imprimir una cadena a través de la USART como método para poder comprobar visualmente que, efectivamente, el microcontrolador había sido correctamente programado. En este caso, al tratarse de la primera parte del proyecto, no se utiliza ningún direccionamiento, de forma que todos los mensajes iban dirigidos siempre a un único nodo sobre el que se estaba realizando la prueba.

La segunda prueba incrementaba el número de nodos de la red, además de mejorar la aplicación de ejemplo. Para este caso se tomó la aplicación de la prueba anterior y se le añadió la posibilidad de enviar la cadena "Ho1a" hacia el bus CAN, con el objetivo de probar la otra funcionalidad de la plataforma, que permite que los nodos inalámbricos envíen mensajes al Servidor. Para ello se hizo uso de la biblioteca CAN que fue desarrollada como parte de este TFG. Además, en el Servidor fue agregada la opción de seleccionar la dirección del nodo que se desea programar. De esta forma, la aplicación de ejemplo para la segunda prueba lo que hace es que cada cierto tiempo imprime la cadena "Ho1a", tanto por la USART como por el bus CAN. De esta forma, es posible comprobar la correcta programación de varios nodos conectados al bus, así como el funcionamiento del eco que realiza el *router* hacia el Servidor de todos los mensajes recibidos a través del bus CAN. A continuación se explica esta segunda prueba más en detalle, comenzando con la interfaz de la aplicación Servidor, que se muestra en la Figura 4.2.



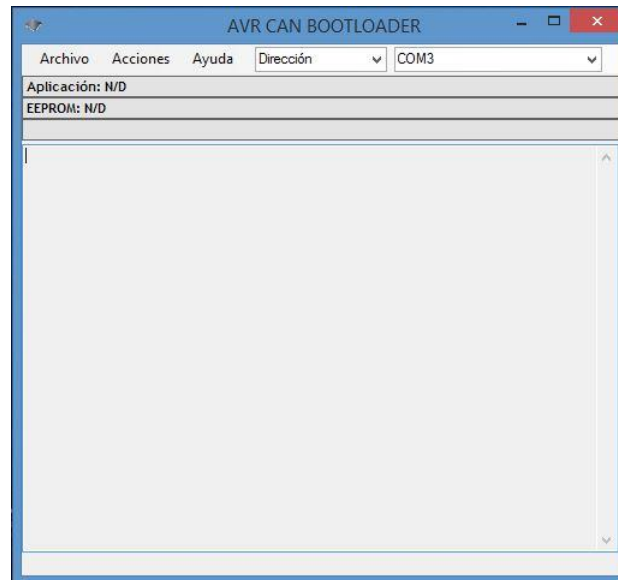


Figura 4.2. Interfaz aplicación Servidor

El proceso para conectarse al *router* se muestra en la Figura 4.3. En primer lugar es necesario seleccionar el puerto serie al cual está conectado el *router*, lo cual se realiza a través del menú desplegable que se encuentra enmarcado en rojo. A continuación, dentro del menú *Acciones*, se selecciona la opción *Conectar*.

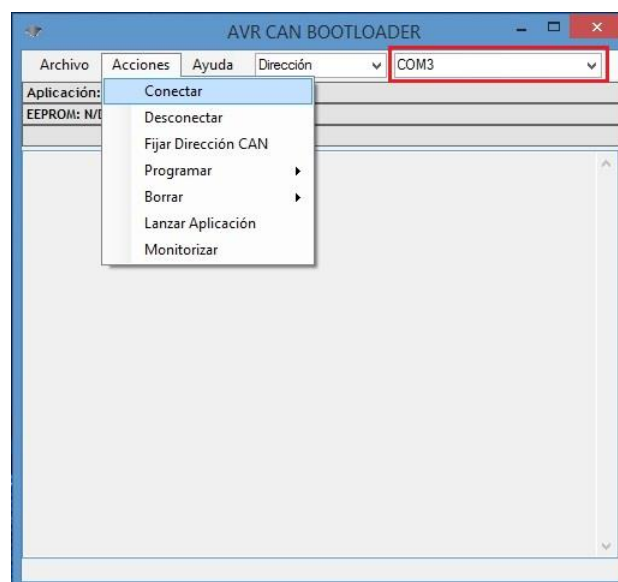


Figura 4.3. Conexión con el router

## Resultados

El siguiente paso consiste en establecer la dirección del nodo que se desea programar, proceso mostrado en la Figura 4.4. En primer lugar es necesario seleccionar la dirección del nodo que se desea programar, lo cual se realiza a través del menú desplegable que se encuentra enmarcado en rojo. A continuación, dentro del menú *Acciones*, se selecciona la opción *Fijar Dirección CAN*.

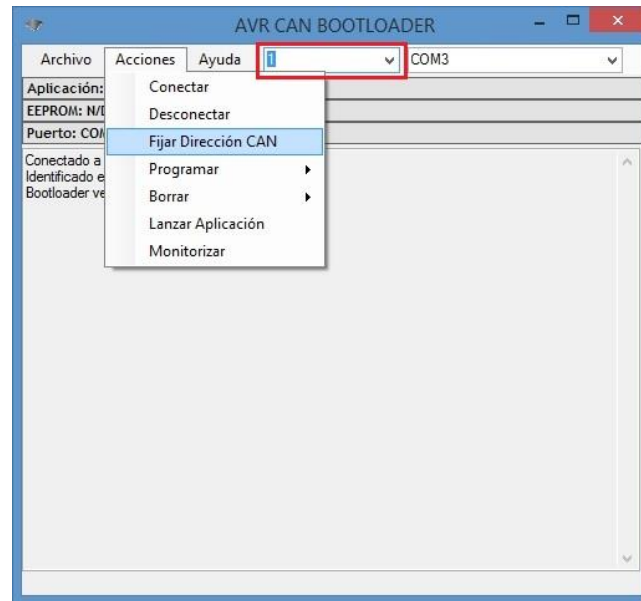


Figura 4.4. Establecer dirección CAN del nodo

Para añadir el código fuente de la aplicación que se desea programar, en primer lugar, dentro del menú *Archivo*, se selecciona la opción *Aplicación* que se encuentra incluida dentro del submenú *Abrir*, tal como se muestra en la Figura 4.5. Esto abrirá una nueva ventana del explorador de archivos, como se puede observar en la Figura 4.6, en la que es necesario buscar el fichero generado por el compilador con la extensión *.hex*.

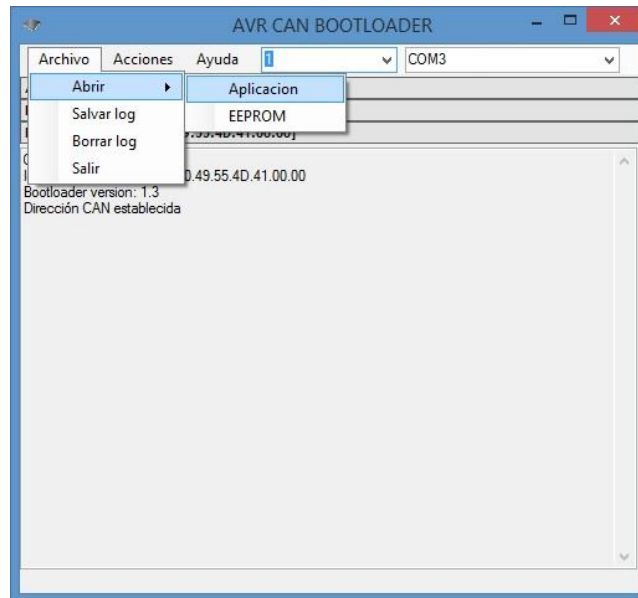


Figura 4.5. Cargar código fuente

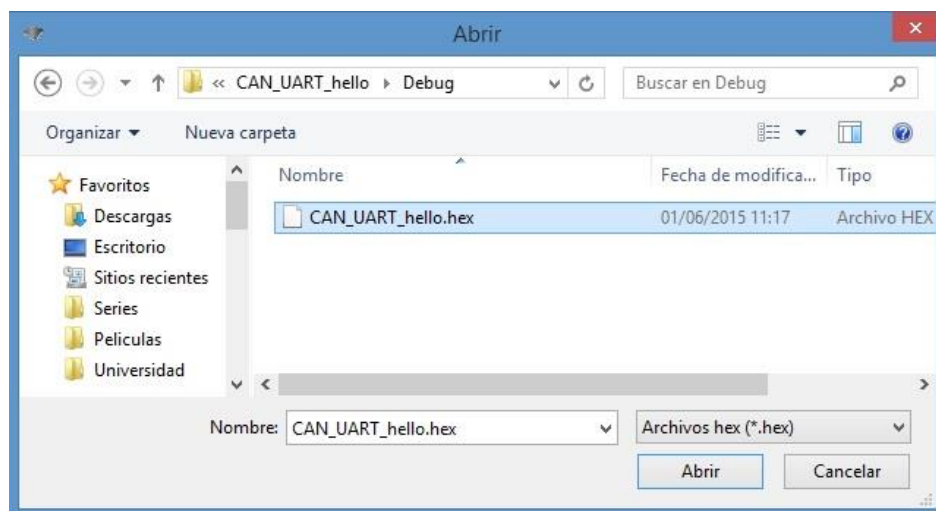


Figura 4.6. Fichero .hex generado por el compilador

En la Figura 4.7, enmarcado en rojo, se encuentra la zona de la aplicación Servidor en la cual se muestran los distintos mensajes de estado. Como se puede observar, entre otras cosas indica a qué puerto está conectada, la dirección MAC del *router*, versión, etc.

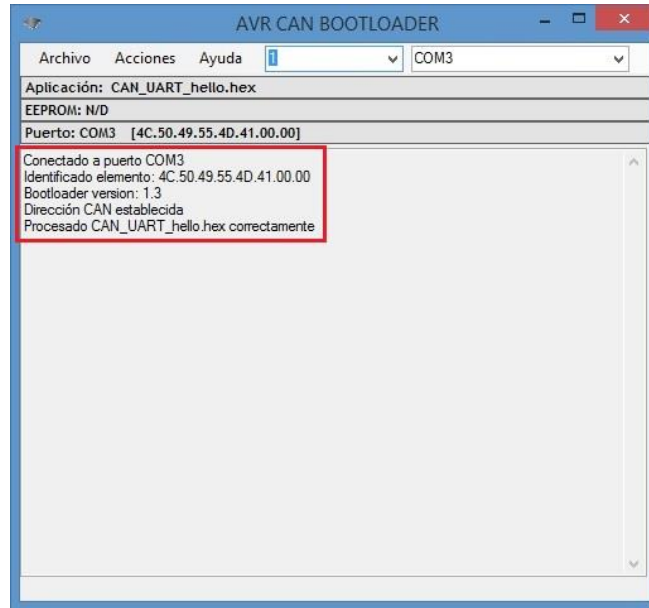


Figura 4.7. Mensajes de estado de la aplicación Servidor

Para programar el nodo con la aplicación que ha sido previamente cargada, es necesario seleccionar la opción *Aplicación*, que está dentro del submenú *Programar*, que a su vez se encuentra dentro del menú *Acciones*. En la Figura 4.8 se muestra este proceso.

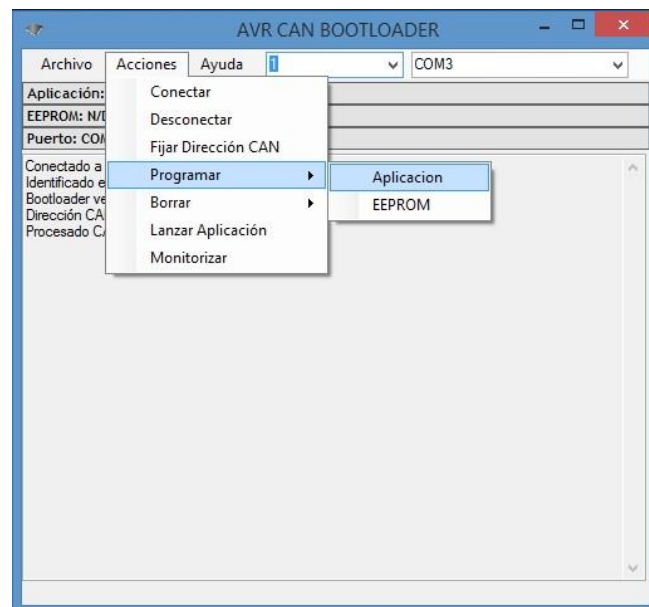


Figura 4.8. Ejecutar programación del nodo

Por otro lado, se ha establecido una conexión con el nodo destino a través de puerto serie, utilizando la aplicación *HyperTerminal*. Como se observa en la Figura 4.9, antes de ejecutar la programación del nodo en la aplicación Servidor, el nodo destino ha enviado a través de la USART un mensaje con la versión de la aplicación *bootloader*.

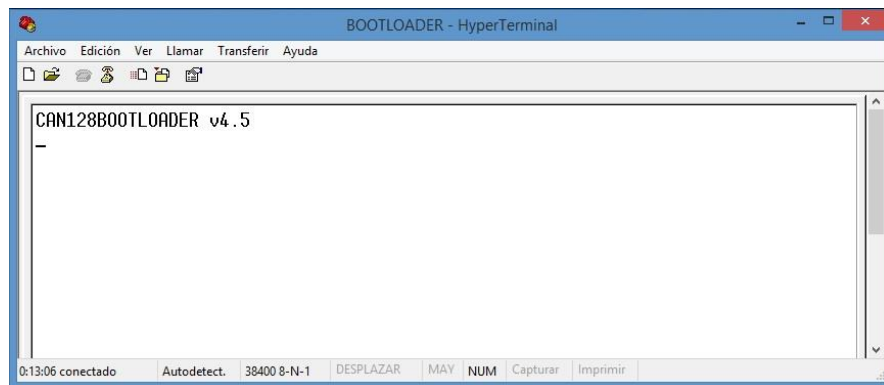


Figura 4.9. Mensaje inicial de la aplicación *bootloader*

En la Figura 4.10 se puede observar la barra que indica el porcentaje completado de la programación del nodo destino.

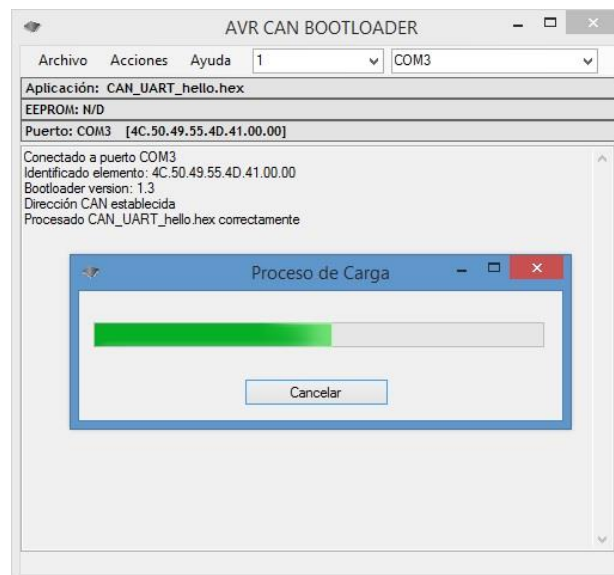


Figura 4.10. Porcentaje completado de la programación del nodo

## Resultados

En la Figura 4.11 se pueden observar los mensajes de confirmación que genera el nodo que está siendo programado para cada comando recibido.

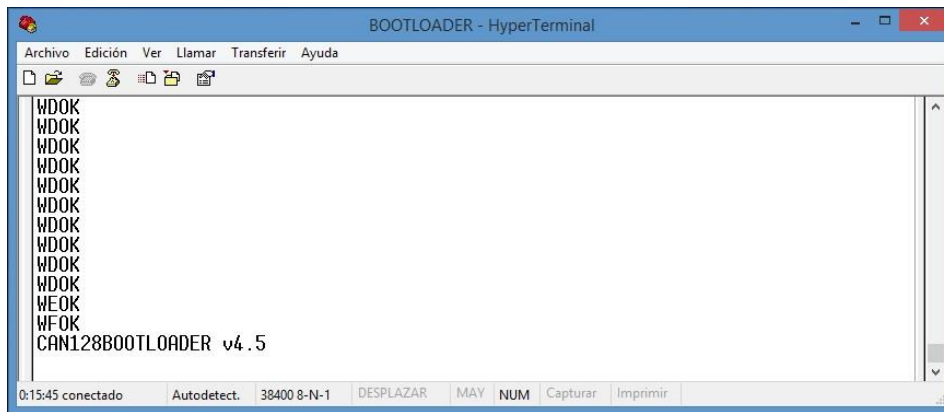


Figura 4.11. Mensajes de confirmación del nodo

Una vez se haya programado el nodo, desde el Servidor es posible lanzar la aplicación de usuario. Para ello es necesario seleccionar la opción *Lanzar Aplicación*, que se encuentra dentro del menú *Acciones*, tal como se muestra en la Figura 4.12.

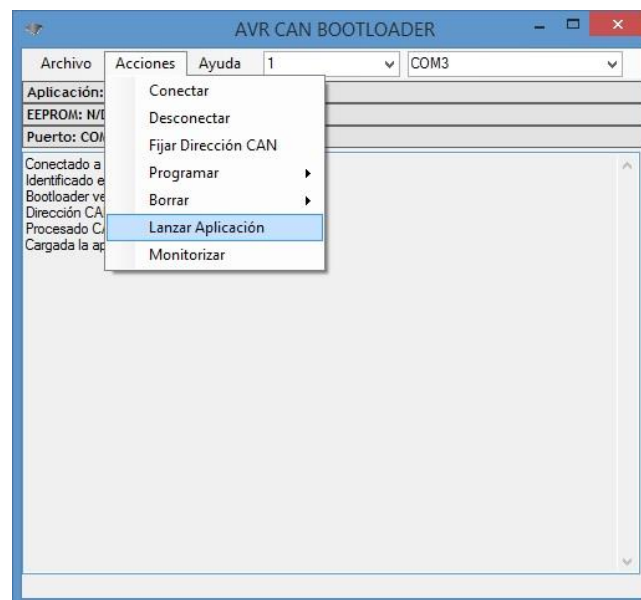



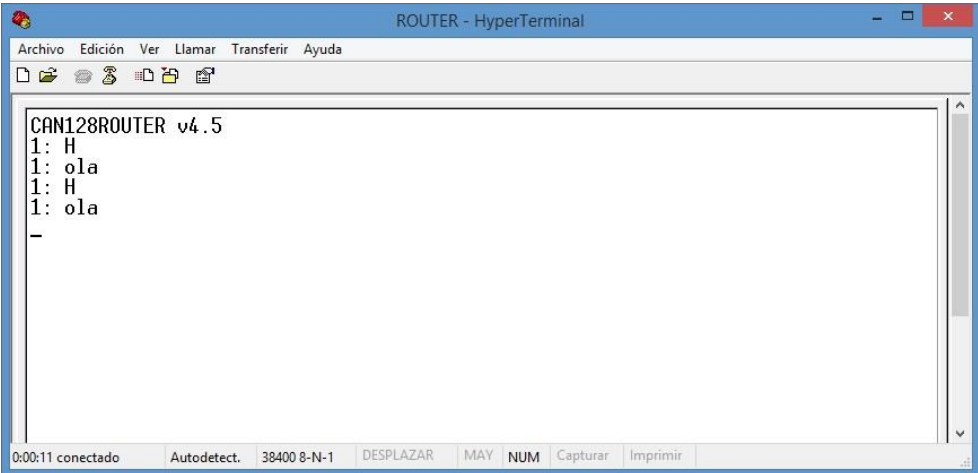
Figura 4.12. Lanzar aplicación de usuario

Como se indicó anteriormente, la aplicación de ejemplo imprime el mensaje "Hola" cada cierto tiempo, tanto por la USART como por el CAN. En el caso del nodo, en la Figura 4.13 se muestra cómo se reciben estos mensajes en el terminal una vez ha sido lanzada la aplicación. Por otro lado, en el caso del *router*, en la Figura 4.14 se muestra cómo además de hacer eco de los mensajes que recibe desde el bus CAN, les añade una cabecera indicando la dirección del nodo que ha enviado el mensaje. En este caso es posible comprobar que tanto la dirección establecida previamente en el Servidor, como la de los mensajes recibidos, coinciden.



```
WDOI
WDOI
WDOI
WDOI
WDOI
CAN128BOOTLOADER v4.5
CAN_UART_hello
Hola
Hola
Hola
Hola
Hola
Hola
```

Figura 4.13. Aplicación de ejemplo en el nodo



```
CAN128ROUTER v4.5
1: H
1: ola
1: H
1: ola
-
```

Figura 4.14. Mensajes recibidos en el router

## | Resultados

A través de estas dos pruebas fue posible comprobar el correcto funcionamiento de la plataforma experimental propuesta. La primera de ellas sirvió para validar la primera parte del proyecto, que era lograr la programación de un nodo a través del bus CAN, y así poder continuar con el desarrollo del mismo. Por otro lado, la segunda prueba validaba el proyecto completo permitiendo validar diferentes aspectos, como son: la posibilidad de seleccionar el nodo inalámbrico que se desea programar, el correcto direccionamiento de estos nodos en el bus, o el funcionamiento del eco del *router* hacia el servidor de todos los mensajes provenientes de los nodos.



# Capítulo 5:

# Conclusiones y Líneas Futuras

## 5.1 Conclusiones.

Con la finalización de este Trabajo Fin de Grado se han conseguido implementar las funciones básicas de programación y monitorización que permiten crear una plataforma experimental que facilitará la realización de pruebas en redes de sensores inalámbricos de una manera mucho más práctica de cara al usuario.

Por un lado, el hecho de que los módulos sensores inalámbricos se basen en el microcontrolador AT90CAN128 ha permitido llevar a cabo un estudio de sus capacidades funcionales, prestando principal atención a aquellas características que han sido de utilidad para la realización de este Trabajo de Fin de Grado. Además se ha podido comprobar que, a pesar de ser un microcontrolador relativamente sencillo, en principio cumple perfectamente con los requisitos necesarios para implementar la plataforma experimental que se pretende desarrollar.

La plataforma a desarrollar permite realizar la programación a distancia y monitorización de los nodos inalámbricos que se encuentran conectados al bus CAN. Para

## | Conclusiones y Líneas Futuras

ello se cuenta con un Servidor central, que es el encargado de leer e interpretar el código fuente desarrollado y generar los comandos que permiten programar dichos nodos. Este Servidor central se comunica con los nodos a través de un *router* que permite direccionar cada uno de ellos, y actúa como interfaz entre el bus CAN y la USART.

Con esta plataforma se conseguirá facilitar la tarea del programador, ya que éste sólo tendrá que centrarse en el desarrollo del *firmware* y no en la implementación de la red de prueba. Además, ofrecerá varias ventajas frente a las redes completamente inalámbricas a la hora de programar los nodos, realizar la depuración, recoger los resultados y en la facilidad para acceder a la red. También será muy escalable, ya que permitirá realizar pruebas iniciales en unos pocos nodos y luego pasar a una red masiva de elementos sin ninguna dificultad.

Una aplicación práctica distinta que se le puede dar a la plataforma es de cara a la docencia y las prácticas de los estudiantes de la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), ya que permitirá al profesor asignarle unos determinados nodos a cada alumno. Estos realizarán sus prácticas en dichos nodos y el profesor podrá analizar los resultados obtenidos por cada alumno, dado que estarán almacenados en el servidor, permitiendo así una evaluación más precisa de las prácticas.

Por otro lado, también ha permitido realizar un acercamiento al entorno AVR de Atmel, siendo una parte importante de ello el aprender a utilizar la herramienta Atmel Studio, la cual es utilizada como entorno de desarrollo para el desarrollo de *firmware* en microcontroladores con núcleo AVR.

Por tanto, se puede considerar que se han cumplido los principales objetivos del presente Trabajo Fin de Grado, al lograr una correcta funcionalidad del sistema desarrollado.

## 5.2 Líneas Futuras.

Como línea futura de trabajo se podría plantear el mejorar y actualizar la aplicación del Servidor. Actualmente esta aplicación sólo permite programar los nodos inalámbricos cuyas direcciones han sido añadidas manualmente. Una mejora podría ser el incorporar una opción que permita llevar a cabo un reconocimiento previo del bus CAN para determinar las direcciones de los nodos que se encuentran conectados al mismo en un momento determinado. Además, la aplicación no procesa los mensajes provenientes de los nodos inalámbricos, pudiendo plantearse el crear un registro (*log*) para cada nodo, en el que se almacenen estos mensajes, añadiéndoles una firma temporal para facilitar su posterior análisis.

Otra ampliación posterior del trabajo desarrollado sería la de crear un servidor web que se comunique con la aplicación. Con esto se lograría crear una interfaz de acceso a la plataforma mucho más intuitiva y ubicua, permitiendo así a los usuarios que utilicen la plataforma, disponer de la comodidad que brinda el poder acceder desde cualquier ubicación física. Además se podría añadir una base de datos en la que los usuarios puedan almacenar su información y resultados para un posterior análisis.



# Referencias Bibliográficas

- [1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks* 38(4), pp. 393-422. 2002.
- [2] J. Yick, B. Mukherjee and D. Ghosal. Wireless sensor network survey. *Computer Networks* 52(12), pp. 2292-2330. 2008.
- [3] Q. Wang and I. Balasingham. "Wireless Sensor Networks-an Introduction". INTECH Open Access Publisher, 2010. [Online]. Available: <http://cdn.intechweb.org/pdfs/12464.pdf>. Accessed on Jul. 8, 2015.
- [4] M. Lodder, G. Halkes and K. Langendoen. "A global-state perspective on sensor network debugging," in *Proc. 5th ACM Workshop on Embedded Networked Sensors*, 2008.
- [5] Z. Chen and K. G. Shin. "Post-deployment performance debugging in wireless sensor networks," in *Real-Time Systems Symposium*, 2009.
- [6] Future Technology Devices International Ltd, "USB to UART interface," FT232R, Datasheet., 2010. [Online]. Available: [http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS\\_FT232R.pdf](http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf). Accessed on Jul. 8, 2015.

## | Referencias Bibliográficas

- [7] Texas Instruments, "CAN transceiver with fast loop times for highly loaded networks," SN65HVD255, Datasheet., 2011. [Online]. Available: <http://www.ti.com/lit/ds/symlink/sn65hvd255.pdf>. Accessed on Jul. 8, 2015.
- [8] Atmel, "8-bit AVR microcontroller with ISP flash and CAN controller," AT90CAN128, Datasheet., 2008. [Online]. Available: <http://www.atmel.com/images/doc7679.pdf>. Accessed on Jul. 8, 2015.
- [9] A. Bogen and V. Wollan. AVR enhanced RISC microcontrollers. *Atmel Technical Document*, 1999.
- [10] G. Myklebust. The AVR microcontroller and C compiler co-design. *ATMEL Development Center, Trondheim, Norway*, 1996.
- [11] C. Kühnel. *AVR RISC Microcontrollers Handbook*. UK: Newnes, 1998.
- [12] M. Di Natale, H. Zeng, P. Giusto and A. Ghosal. *Understanding and using the Controller Area Network Communication Protocol: Theory and Practice*. USA: Springer, 2012.
- [13] W. Voss. *A Comprehensive Guide to Controller Area Network*. USA: Copperhill Media, 2005.

# Pliego de Condiciones

El presente pliego de condiciones tiene como objeto fijar las condiciones bajo las que se ha desarrollado y se rige este Trabajo Fin de Grado.

## PL.1 Condiciones Hardware.

Los elementos *hardware* empleados durante el desarrollo de este Trabajo Fin de Grado se detallan en la Tabla PL.1.

Equipo	Modelo
Ordenador de trabajo	Asus A53S
Emulador AVR	Atmel JTAGICE3
Módulos sensores inalámbricos experimentales basados en el $\mu$ C AT90CAN128	

Tabla PL.1. Elementos hardware

## PL.2 Condiciones Software.

Los elementos *software* empleados en el desarrollo de este TFG se recogen en la Tabla PL.2.

Concepto	Versión	Fabricante
Sistema Operativo	Microsoft Windows 8.1 Pro x64	Microsoft
Entorno de Desarrollo del Firmware	Atmel Studio 6 Versión:6.2.1548 Service Pack 2	Atmel
Entorno de Desarrollo de la Aplicación Servidor	Microsoft Visual Studio 2010	Microsoft
Editor de Textos	Microsoft Office 2013	Microsoft

Tabla PL.2. Elementos software





# Presupuesto

Siguiendo las recomendaciones del COITT (Colegio Oficial de Ingenieros Técnicos de Telecomunicación), el presupuesto ha sido desglosado en varias secciones. En estas secciones se han separado los distintos costes orientativos asociados al desarrollo del proyecto. Estos costes se dividen en:

1. Recursos materiales.
2. Trabajo tarifado por tiempo empleado.
3. Material fungible.
4. Costes de redacción del Trabajo Fin de Grado.
5. Derechos de visado del COITT.
6. Costes de tramitación y envío.
7. Aplicación de impuestos.

## P.1 Recursos materiales.

Para la realización de este Trabajo Fin de Grado ha sido necesario utilizar tanto recursos *hardware* como recursos *software*. La amortización de estos recursos se calcula sobre el tiempo de vida útil del mismo.

### P.1.1 Recursos Hardware.

Para la realización del presente TFG se han empleado las herramientas *hardware* que se detallan en la Tabla P.1.

Recurso	Valor de adquisición	Vida útil (meses)	Coste mensual	Tiempo de uso (meses)	Importe
<b>Ordenador personal Asus A53S</b>	500 €	48	10,42 €	4	41,68 €
<b>Módulos Sensores Inalámbricos</b>	150 € x 2	-	-	-	300,00 €
<b>Emulador AVR – JTAGICE3</b>	90 €	-	-	-	90,00 €
<b>Impresora</b>	50 €	48	1,04 €	4	4,16 €
<b>TOTAL</b>					<b>435,84 €</b>

*Tabla P.1. Recursos hardware*

### P.1.2 Recursos Software.

Para la realización del presente TFG se han empleado herramientas software que se detallan en la Tabla P.2.

Recurso	Valor de adquisición	Vida útil (meses)	Coste mensual	Tiempo de uso (meses)	Importe
<b>Microsoft Office 2013</b>	269 €	48	5,06 €	4	20,24 €
<b>Microsoft Visual Studio 2010</b>	647,00 €	48	13,48 €	4	53,92 €
<b>Atmel Studio 6.2</b>	0 €	-	-	-	0 €
<b>TOTAL</b>					<b>74,16 €</b>

*Tabla P.2. Recursos software*

Los costes totales asociados a los recursos materiales libres de impuestos ascienden a quinientos diez euros (**510,00 €**).

## P.2 Trabajo tarifado por tiempo empleado.

El importe de las horas de trabajo empleadas para la realización de un proyecto se calcula siguiendo las recomendaciones del COITT a partir de la siguiente ecuación:

$$\text{Honorarios (€)} = H_n * 58 + H_e * 63$$

Donde:

- $H_n$  son las horas normales trabajadas (dentro de la jornada laboral).
- $H_e$  son las horas especiales.

Estos honorarios tendrán una reducción variable en función del número de horas empleadas de acuerdo con la Tabla P.3.

Horas empleadas	Factor de corrección $C_t$
Hasta 36 horas	1.00
Desde 36 a 72 horas	0.90
Desde 72 a 108 horas	0.80
Desde 108 a 144 horas	0.70
Desde 144 a 180 horas	0.65
Desde 180 a 360 horas	0.60
Desde 360 a 510 horas	0.55

*Tabla P.3. Factor de corrección según las horas trabajadas*

En este Trabajo Fin de Grado se han invertido 300 horas en las tareas de preparación, desarrollo y documentación necesarias para la elaboración del mismo. Por

## | Presupuesto

tanto el factor de corrección correspondiente es  $C_t = 0,60$  y el resultado de aplicar la ecuación anterior es el siguiente:

$$\text{Honorarios (€)} = (300 * 58 + 0 * 63) * 0,6 = 10.440,00$$

Los honorarios totales por tiempo dedicado libres de impuestos ascienden a diez mil cuatrocientos cuarenta euros (**10.440,00 €**).

### P.3 Material fungible.

Los costes relacionados con el material fungible utilizado en la realización de este TFG se detallan en la Tabla P.4.

Material	Coste
Materiales de papelería	10,00 €
CD-ROM	5,00 €
Impresión	40,00 €
Encuadernación	5,00 €
<b>TOTAL</b>	<b>60,00 €</b>

*Tabla P.4. Costes material fungible*

Los costes totales asociados al material fungible libres de impuestos ascienden a sesenta euros (**60,00 €**).

## P.4 Costes de redacción del Trabajo Fin de Grado.

El importe de la redacción del proyecto viene definido por la siguiente ecuación:

$$R = 0,07 * P * C_n$$

Donde:

- $P$  es el presupuesto del proyecto.
- $C_n$  es el coeficiente de ponderación en función del presupuesto.

El presupuesto acumulado hasta el momento se muestra en la Tabla P.5.

Concepto	Coste
Recursos materiales	510,00 €
Honorarios por tiempo empleado	10.440,00 €
Material fungible	60,00 €
<b>TOTAL</b>	<b>11.010,00 €</b>

*Tabla P.5. Presupuesto acumulado*

El presupuesto acumulado hasta el momento es de 11.010,00 €. Por su parte, el coeficiente de ponderación  $C_n$  tiene valor unitario, ya que se trata de un proyecto que no supera los 30.050,00 € de coste.

$$R = 0,07 * 11.010,00 * 1,00 = 770,70$$

## | Presupuesto

Los costes de redacción del presente TFG libres de impuestos ascienden a setecientos setenta euros con setenta céntimos (**770,70 €**).

## P.5 Derechos de visado del COITT.

Los gastos de visado del COITT se tarifican mediante la siguiente ecuación:

$$V = 0,006 * P * C_v$$

Donde:

- $P$  es el presupuesto del proyecto.
- $C_v$  es el coeficiente reductor en función del presupuesto del proyecto.

Teniendo en cuenta el presupuesto acumulado hasta el momento asciende a 11.780,70 €, y que el valor del coeficiente  $C_v$  es de valor unidad ya que se trata de un proyecto que no supera los 30.050,00 € de coste, se tiene:

$$V = 0,006 * 11.780,70 * 1,00 = 70,68$$

Los costes de obtención de los derechos de visado del COITT del presente TFG libres de impuestos ascienden a setenta euros con sesenta y ocho céntimos (**70,68 €**).

## P.6 Costes de tramitación y envío.

Los gastos de tramitación y envío están fijados en 6,01 euros.

## P.7 Aplicación de impuestos.

Para la actividad económica del presente TFG el valor del Impuesto General Indirecto Canario (I.G.I.C.) graba el presupuesto con un 7%. El coste total del proyecto se desglosa en la Tabla P.6.

Concepto	Coste
Recursos materiales	510,00 €
Honorarios por tiempo empleado	10.440,00 €
Material fungible	60,00 €
Costes de redacción	770,70 €
Derechos de visado del COITT	70,68 €
Costes de tramitación y envío	6,01 €
Subtotal	11.857,39 €
I.G.I.C. (7%)	830,02 €
<b>TOTAL</b>	<b>12.687,41 €</b>

*Tabla P.6. Coste total del Trabajo Fin de Grado*

El presupuesto final del presente Trabajo Fin de Grado, titulado “Desarrollo del *Firmware* de Control de los Nodos de una Plataforma Experimental de WSN”, asciende a un total de doce mil seiscientos ochenta y siete euros con cuarenta y un céntimos (**12.687,41 €**).

Las Palmas de Gran Canaria a 14 de julio de 2015.

Fdo.: Ricardo Antonio Rios Peña.





# Anexo I.

## Contenido del CD-ROM

Adjunto a la memoria de este Trabajo Fin de Grado se encuentra disponible un CD-ROM. En este anexo se describe la estructura del mismo.

### A.I.I Estructura del CD-ROM.

El contenido y la estructura del CD-ROM que se adjunta es el siguiente:

- Directorio "*Memoria*". Memoria del presente TFG en formato PDF.
- Directorio "*CAN128ROUTER*". Aplicación *router* desarrollada.
- Directorio "*CAN128BOOTLOADER*". Aplicación *bootloader* desarrollada.
- Directorio "*CAN128LIB*". Biblioteca CAN desarrollada.
- Directorio "*Summary*". Resumen del TFG en inglés.