Tesina del Master en Automática e Informática Industrial: Implementación de un sistema estandarizado de comunicación bajo protocolo *ZigBee*

Autor
Salvador Company Andrés
salvador.company.andres@gmail.com

Directores
Ángel Valera Fernández
Marina Vallés Miquel

20 de junio de 2012

Índice general

1. Introducción **4**

1.1. Objeto de la tesina 4

1.2. Estructura del documento 5

2. Descripción de las diferentes tecnologías de comunicación inalámbricas **7**

2.1. Descripción de las diferentes alternativas inalámbricas 7

2.1.1. Norma IEEE 802.15.4 como estándar de las WPAN: ZigBee 7

2.1.2. Norma IEEE 802.15.1 como estándar de las WPAN: Bluetooth 9

2.1.3. Norma IEEE 802.11: Wi-fi 10

2.2. Comparación de las diferentes alternativas inalámbricas 11

2.3. Elementos fundamentales en el protocolo *ZigBee* 11

2.3.1. La *ZigBee* Alliance 11

2.3.2. Frecuencia de operación, DSSS y mecanismos de acceso 13

2.3.3. Pila *ZigBee* y protocolos asociados 13

2.3.4. Tipos de dispositivos y topologías de red 15

2.3.5. Algunas aplicaciones prácticas 17

3. Hardware para la implementación de la comunicación bajo *ZigBee* **18**

3.1. Consideraciones generales 18

3.2. Descripción del *SoC* CC2530 y del CC2531 19

3.2.1. CPU y memoria 20

3.2.2. Relojes y gestión energética 20

3.2.3. Periféricos 20

3.2.4. Radio 22

3.3. Descripción del CC2350EM 22

3.4. Descripción del SmartRF05 Evaluation Board 23

4. Software en la implementación bajo *ZigBee* para los equipos de Texas Instruments 26
4.1. Descripción del entorno de desarrollo: IAR 26
4.2. Estructura de un proyecto de software en IAR 27
4.3. Z-Stack de Texas instruments para CC2530 y CC2531: Application Framework . 29
4.4. Otras librerías importantes 31
4.4.1. HAL 31
4.4.2. OSAL 32
4.5. Compilación y programación 33

5. Estandarización de la trama *ZigBee* 35
5.1. Justificación de la necesidad de estandarizar la trama 35
5.2. Estructura propuesta 35
5.3. Comandos e interpretación de las diferentes versiones de la trama 36

6. Implementación de un puente UART/*ZigBee* 38
6.1. Hardware empleado 38
6.2. Descripción del software empleado 38
6.3. Terminal Serie en PC 43
6.3.1. Dos procesos independientes para lectura/escritura 45
6.3.2. Un único proceso de lectura/escritura/toma de decisiones implementado con hilos y secciones críticas 47
6.4. Funcionamiento del sistema 50
6.5. Verificación del funcionamiento del sistema 51

7. Aplicación al control de *Robotino* 53
7.1. Descripción de *Robotino* 53
7.2. Conexión física entre el *SmartRF05EB* y *Robotino* 55
7.3. Secuencia de operaciones 55
7.4. Resultado de la ejecución 56

8. Implementación final propuesta 57
8.1. Justificación para una implementación definitiva distinta 57
8.2. Adaptación del hardware 57
8.3. Adaptación del software 58

9. Pasos siguientes 60

10. Conclusiones **61**

A. Software propuesto sobre CC2530 y SmartRF05 **63**

B. Software de control propuesto para enviar y recibir mensajes y ejercer tareas de control **89**

B.1. Listados para el ejecutable *escribiryenviarconhilos* 89

B.2. Listados para la librería *zigbeeti* 98

Capítulo 1

Introducción

1.1. Objeto de la tesina

El presente documento aborda un problema común en la comunicación entre equipos y en particular entre robots: la heterogeneidad de los protocolos y tramas de comunicación que se encuentran en diversos equipos. Robots que se comunican utilizando redes Wifi pueden tener que convivir con robots que se comunican empleando Bluetooth o cualquier otro protocolo que permita comunicación inalámbrica. Además, las tramas que se transmiten pueden tener estructuras internas, de forma que la comunicación entre todos estos equipos exige un conocimiento de los protocolos y de las tramas empleadas que complican sobremanera su uso conjunto, y derivan gran parte del esfuerzo técnico a resolver problemas que podrían tener una solución simple si se hubiera realizado una tarea previa de estandarización.

Ante esta situación el objeto de este documento es presentar una solución simple basada en *ZigBee*, protocolo inalámbrico orientado al control que permite enviar y recibir órdenes con un consumo energético sustancialmente más bajo que el de otros sistemas de comunicación inalámbricos, como Wifi y Bluetooth. El tema energético es muy relevante sobre todo en aquellos elementos que no están constantemente conectados a una red eléctrica, como los robots móviles, o para aquellos equipos que, por su naturaleza, se encuentran en espacios abiertos alejados de la posibilidad de una fuente de energía fiable. Sobre *ZigBee* se ha definido una trama de 32 octetos que incluye en diferentes posiciones estandarizadas información relevante para que la citada trama se pueda procesar por un *System On Chip* (*SoC*). Esta trama se supone generada o bien por un computador de control o por el microcontrolador embarcado en el robot que intenta comunicarse con el computador de control [1] y se supone recibida por el *SoC* vía UART. Al revés, cuando el *SoC* recibe un mensaje *ZigBee* también se espera que contenga una trama estándar de 32 octetos que se procesará enviándola via UART al microcontrolador de control o bien al computador de control.

Encontrar una solución al problema anterior ha implicado una serie de pasos tanto en la selección del hardware como en la programación del mismo, así como en la definición de un estándar de comandos a implementar sobre las tramas. Estos pasos han sido los siguientes:

- Estudio exhaustivo de la arquitectura de harware de control y comunicación de los robots

[1] Aunque también podría comunicarse con otros robots, una vez tuviera acceso al directorio de las direcciones disponibles via coordinador

móviles típicos, segregando entre el módulo de control y el módulo de comunicaciones. En particular se estudio con mucho detalle los microcontroladores de MicroChip[2] y los SoCs de comunicación de Texas Instruments[3], así como la arquitectura de drivers para el control de las ruedas y monitorización de los sensores. Aunque el presente documento se centra en el CC2530 de Texas Instruments, en las referencias [2] y [1] se puede encontrar la descripción de los dsPIC que se emplean en el diseño de pequeños robots, y en [11] y [13] se describe la librería en C para los dispositivos de 16 bits de Microchip y el RTOS *Erika*, respectivamente

- Asimismo se estudió de forma exhaustiva las herramientas de programación tanto de los dsPIC32 como de los CC2530[4].
- Programación en C del código empleado como puente *ZigBee* / UART y UART / *ZigBee* de acuerdo a un determinado formato para la trama transmitida tanto par el SoC empleado como Coordinador como para los SoC empleados como End Devices sobre el kit de desarrollo. Una buena referencia sobre el lenguaje es la referencia [22].
- Propuesta de estandarización de una colección de tramas que comprendan todos los comandos necesarios en la comunicación entre robots móviles.
- Descripción del hardware necesario para la implementación general en cualquier sistema que reciba vía UART. En la referencia [16] se encuentra buena descripción genérica de los puertos serie.

Cada uno de estos pasos se describe en el presente documento.

El objetivo final ha sido el de ofrecer una solución comprensiva tanto a nivel de software, hardware y estandarización de la trama, y se ha propuesto como siguientes pasos la implementación en los robots o equipos que necesiten incorporar una comunicación inalambrica para su correcto funcionamiento.

1.2. Estructura del documento

El presente documento se estructura en las siguientes partes:

- En primer lugar se describen las diferentes tecnologías inalámbricas candidatas para resolver el problema de comunicación. De ellas se selecciona una: *ZigBee*.
- Se realiza una descripción pormenorizada de las características del protocolo *ZigBee*.
- Posteriormente se revisa el hardware disponible para implementar la comunicación *ZigBee* que se encuentra en el mercado, y en particular las soluciones de Texas Instruments.
- Se describe la programación del puente en el harware seleccionado, describiendo previamente el entorno de desarrollo y las librerías disponibles, en particular el Z-Stack, que corresponde a la librería *ZigBee* propietaria de Texas Instruments.

[2] Los dsPIC32
[3] Los CC2530 y CC2531
[4] Con MPLAB bajo sistema operativo Erika y IAR con OSAL propio de Z-Stack

- Se describe la lista estandarizada de comandos a implementar sobre las tramas.
- Se propone la implementación sobre entornos reales.

Capítulo 2

Descripción de las diferentes tecnologías de comunicación inalámbricas

2.1. Descripción de las diferentes alternativas inalámbricas

2.1.1. Norma IEEE 802.15.4 como estándar de las WPAN: ZigBee

IEEE 802.15 es el undécimoquinto grupo de trabajo de la norma IEEE 802. Está especializado en WPANs (Wireless Personal Area Networks). Incluye siete subgrupos. En particular el cuarto grupo cuarto trabaja en el estándar que determina el medio físico (PHY) y el acceso al medio (MAC) para redes inalámbricas personales con bajo ancho de banda.

Se busca sobre todo la definición de redes de nodos/sensores de bajo coste y bajo consumo, y con poca infraestructura.

El modelo básico describe un rango de comunicación con un ancho de banda de 250kbits/segundo. Otras características incluyen la reserva de slots para incluir potenciales aplicaciones de tiempo real, protocolo CSMA/CA para garantizar un acceso al medio sin colisiones y soporte para comunicaciones seguras. El protocolo propuesto describe el nivel físico de las capas del modelo OSI y Nivel de enlace de datos (MAC). El Nivel de enlace de datos LLC está definido por el IEEE 802.2, tal y como se describe en la figura 2.1.

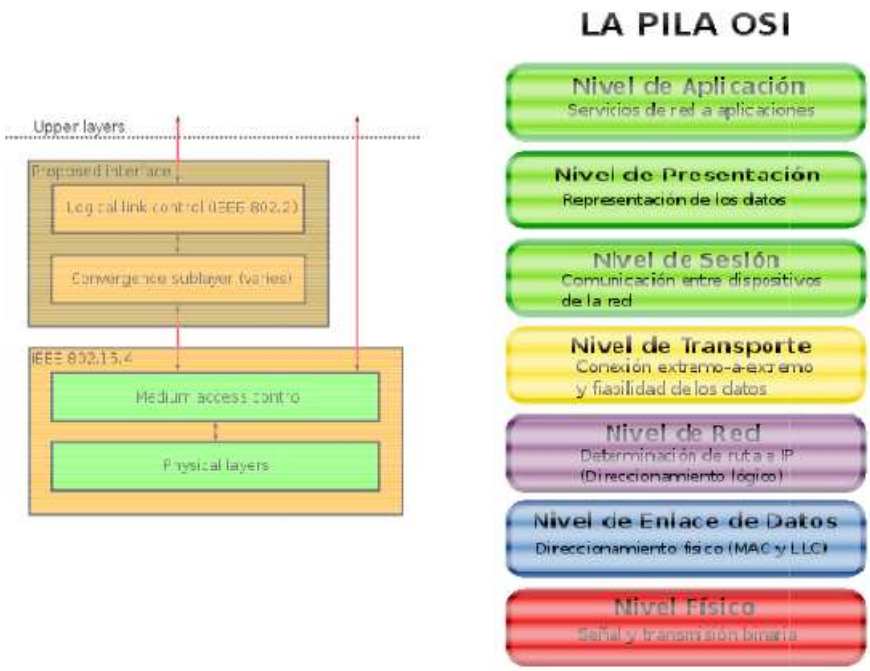

Figura 2.1: Esquema del regulador a estudiar

La capa física (PHY) describe el servicio de transmisión de datos. La capa física controla el receptor/emisor RF y gestiona la selección del canal y la gestión de energía y gestión de la señal. Opera en alguna de las tres bandas siguientes de frecuencia:

- 868.0-868.6 MHz: en Europa (un canal). Versiones del IEEE802.15.4-2003 y 2006.
- 902-928 MHz: NorteAmérica (hasta DIEZ canales, IEEE802.15.4-2003; hasta TREINTA canales, IEEE802.15.4-2006).
- 2400-2483.5 MHz: Internacional (hasta DIECISÉIS CANALES, IEEE802.15.4-2003 y 2006).

Las distancias a las que los dispositivos ZigBee pueden transmitir varían entre 10m y 75m, pudiendo alcanzar 1500m para ZigBee pro.

La versión IEEE 802.15.4-2003 especificaba dos capas físicas basadas en técnicas DSSS, una trabajando en las bandas 868-915 MHz con ancho de banda entre 20 y 40 kbps, y una en la frecuencia 2450 MHz con un ancho de banda de 250 kbps. La revisión IEEE 802.15.4-2006 aumenta el ancho de banda de las frecuencias centradas en 868 MHz y 915MHz, elevándolas hasta 100 kbps y 250 kbps, respectivamente. Además, define cuatro capas físicas en función del método de modulación empleado. Estas capas PHY fueron ampliadas a seis en agosto de 2007, y de nuevo ampliadas en abril de 2009.

La capa de acceso al medio (MAC) describe la transmisión de tramas MAC a través de la capa PHY descrita anteriormente. Ofrece una interfaz para gestionar el acceso a medio físico y autorreparación (beaconing). También verifica las tramas, garantiza slots y distribuye las asociaciones entre nodos. El acceso al medio se resuelve utilizando un protocolo CSMA/CA.

Otras capas no están definidas en este estándar. Sin embargo, existen especificaciones que se construyen a partir de lo especificado por la IEEE 802.15.4 y que generan una serie de

soluciones distintas. Una de de estas implementaciones es ZigBee. Otras implementaciones utilizan parte de este estándar, como por ejemplo el sistema operativo TinyOS.

2.1.2. Norma IEEE 802.15.1 como estándar de las WPAN: Bluetooth

La relación de la tecnología Bluetooth con las normas IEEE es menos directa que la de *ZigBee*. Inicialmente se generaron las especificaciones para Bluetooth siguiendo las instrucciones generadas a partir del Bluetooth SIG, y la especificación Bluetooth 1.1 fue ratificada como estándar por la IEEE generando la IEEE 802.15.1-2002. Bluetooth 1.2 se convirtió en el estándar IEEE 802.15.1-2005. Posteriormente, la IEEE ha roto su relación con el Bluetooth SIG, de forma que las siguientes especificaciones de Bluetooth no son estándares de la IEEE, sino especificaciones generadas por la Bluetooth SIG. La última especificación es la Bluetooth 4.0.

En cualquier caso, las características de Bluetooth son diferentes de las de *ZigBee*, y en particular de los protocolos y tecnologías que se basan en el estándar IEEE 802.15.4. Bluetooth se define como una arquitectura que incluye protocolos principales, protocolos para reemplazar cables y protocolos para reemplazar hilos telefónicos, entre otros. Los protocolos obligatorios son LMP (Link Management Protocol), L2CAP (Logical Link Control Adaptation Protocol) y SDP (Service Discovery Protocol). Además, HCI (Host/Controller Interface) y RFCOMM (Serial Port Emulation) son soportados por la inmensa mayoría de los dispositivos. También se soportan una serie de protocolos adoptados de otros estándares, como PPP (Point-to-Point Protocol) y el TCP/IP, entre otros.

Para la conexión cualquier dispositivo Bluetooth en modo “discoverable”, al recibir una petición, envía su nombre, clase, servicios disponibles e información técnica sobre características del dispositivo y especificación Bluetooth. Cualquier dispositivo puede realizar un barrido para encontrar otros dispositivos, y cualquier dispositivo puede ser configurado para responder a las peticiones. Sin embargo, si el dispositivo que intenta conectarse conoce la dirección -direcciones de 48 bits, aunque las direcciones no se muestran en las respuestas, sino nombres o identificadores- del dispositivo, siempre responde a peticiones de conexión directas y transmite la información enumerada en el párrafo anterior. En cualquier caso el uso de los servicios que ofrece un determinado dispositivo requiere la aceptación por parte del dispositivo que ofrece los citados servicios. De hecho, algunos dispositivos pueden estar conectados a un único dispositivo en un determinado momento, y esto les impide conectarse a otros dispositivos y responder a peticiones hasta que se desconecta.

Muchos de los servicios ofrecidos en una red Bluetooth pueden poner en peligro datos privados o permitir al dispositivo que se conecta el controlar el dispositivo Bluetooth. Es por ello necesario controlar a qué dispositivos se les permite conectarse a un determinado dispositivo Bluetooth. A la vez, es interesante permitir a los dispositivos Bluetooth, cuando se haya validado la seguridad de la conexión, el conectarse automáticamente en cuanto esto sea posible. Para resolver esta aparente contradicción Bluetooth utiliza el “pairing” (emparejado). El “pairing” se inicia (generalmente de forma manual) por el usuario de un dispositivo, haciendo de esta forma la posibilidad de enlace visible a otros dispositivos. De hecho, dos dispositivos deben estar emparejados para comunicarse. Una vez que el emparejado se ha establecido, éste es recordado por los dispositivos, y se ha de eliminar cuando se quiera que la asociación deje de ser válida.

Durante el proceso de emparejado, los dos dispositivos involucrados establecen una relación creando un secreto compartido (clave compartida). Una vez este secreto ha sido almacenado

por ambos dispositivos, se dice que los dispositivos están emparejados. Cuando un dispositivo que quiere comunicarse únicamente con un dispositivo con el que está emparejado puede validar criptográficamente la identidad del otro dispositivo, y de esta forma tener la certeza de que es realmente el dispositivo con el que se emparejó. Una vez se ha generado y almacenado la clave compartida, la transmisión de información se puede encriptar.

En cuanto a las frecuencias utilizadas, Buetooth opera en la banda entre 2.402 GHz a 2.48 GHz. El protocolo Bluetooth divide la banda en 79 canales de 1 MHz de amplitud y cambia de canales hasta 1600 veces por segundo. En cuanto a la velocidad, las versiones 1.1 y la 1.2 alcanzan velocidades de 723.1 Kbps. La versión 2.0 alcanza los 2.1 Mbps. Los dispositivos de Clase 1 tienen un alcance de hasta 100m, mientras que los de Clase 2 tienen un alcance de hasta 10m. A mayor alcance, mayor consumo.

2.1.3. Norma IEEE 802.11: Wi-fi

La tecnología Wi-Fi se desarrolla a partir del estándar IEEE 802.11. Aunque la IEEE desarrolla el estándar en el que se basa la tecnología Wi-Fi, esta organización no se encarga de verificar los equipos que afirman cumplirlo. Para ello se creó en 1999 la Wi-Fi Alliance, para establecer e implementar equipos y para promover la tecnología WLAN. En 2010 la Wi-Fi Alliance cuenta con más de 375 compañías. Aquellas compañías miembro de la Wi-Fi Alliance cuyos productos superan el proceso de certificación obtienen el derecho de etiquetar sus productos con el logo Wi-Fi. En particular, el proceso de certificación requiere que se verifique el cumplimiento del estándar IEEE 802.11, de los estándares de seguridad WPA y WPA2 y del estándar de autenticación EAP.

Los usos de Wi-Fi están orientados a la geranación de WLANs que permitan acceso a redes o internet a ciertos dispositivos o equipos.

En cuanto al uso de las frecuencias requeridas para la comunicación, éstas no operan de forma consistente en todo el mundo. En la mayor parte de Europa se permiten dos canales más de los que permiten en USA en la banda de 2.4GHz (1-13 en lugar de 1-11). Japón, de hecho, tiene uno más (1-14).

Las redes Wi-Fi tienen un rango limitado. Un router típico usando 802.11b, o g, con una antena estándar, tendría un alcance de una 30m en interiores y de unos 100m en exteriores. El 802.11n tiene un alcance de hasta 200m. Estos requerimientos implican un requerimiento de energía relativamente alto. Además, las aplicaciones Wi-Fi disponen de seguridad implementada a través del estándar de seguridad WEP. El cifrado WPA y WPA2 intentan mejorar en este punto.

Los dispositivos más comunes en una red Wi-fi son los siguientes:

- WAP (Wireless Access Point): conecta un grupo de dispositivos inalámbricos con una red LAN adjacente.
- Adaptadores de Red (Wireless Adapters): permite a los equipos conectarse a una red inalámbrica Wi-fi. Para ello utilizan interconexiones internas como PCI, miniPCI y USB, entre otros.
- Routers Inalámbricos: integran WAP, Ethernet, switch y aplicaciones que suminstran enrutado IP y servicio DNS, entre otros servicios.

- Puentes de Red: conectan una red cableada a una inalámbrica.
- Repetidores: aumentan el alcance de una red existente.

2.2. Comparación de las diferentes alternativas inalámbricas

En el cuadro 2.1 se puede ver un resumen de los puntos más importantes para las tres tecnologías consideradas.

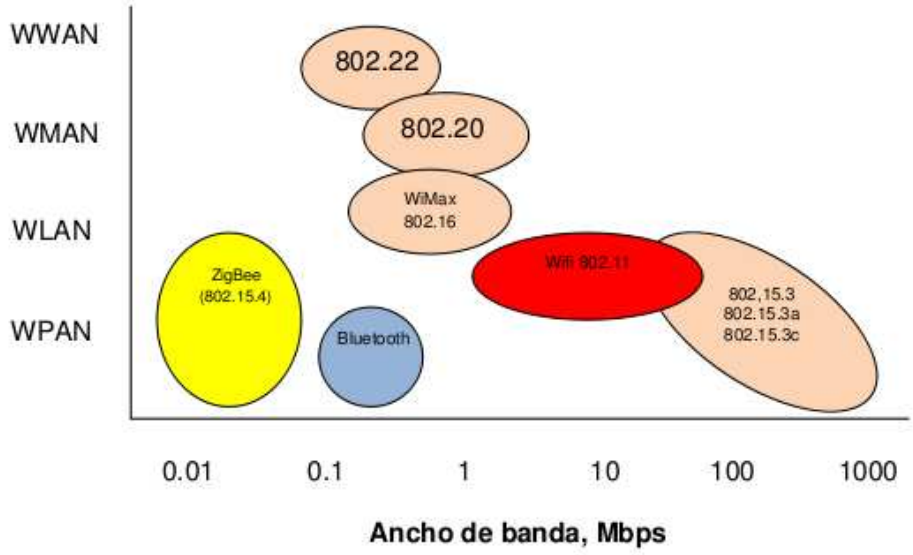

Figura 2.2: Comparación de las diferentes tecnologías inalámbricas

Se puede observar en el cuadro 2.1 y en la figura 2.2 que en aquellas aplicaciones móviles con restricciones importantes de carga en batería *ZigBee* ofrece un compromiso ideal entre ancho de banda, consumo energético y alcance. De hecho, la posibilidad de configurar un dispositivo *ZigBee* como coordinador, end device o router permite que el alcance teórico de la red se multiplique sustancialmente, puesto que los dispositivos *router* pueden funcionar como end devices a la vez que actúan como repetidores de señal en busca de destinatario.

2.3. Elementos fundamentales en el protocolo *ZigBee*

2.3.1. La *ZigBee* Alliance

La promoción de los dispositivos *ZigBee* así como la supervisión del cumplimiento de los estándares de la tecnología se realizan desde la *ZigBee Alliance* (similar en concepción a la Wifi Alliance y a la Bluetooth SIG). Esta asociación sin ánimo de lucro (al menos directo, claro) reúne a una serie de compañías en diferentes niveles de pertenencia (promoters, participants y adopters) que tienen acceso a los estándares y a los procesos de definición de las mejoras y cambios que conformarán la evolución del estándar de la tecnología. Algunos de los promoters son Schneider Electric, Texas Instruments y Philips.

Cuadro 2.1: Comparación de las tecnologías inalámbricas consideradas

	ZigBee	**Bluetooth**	**Wi-fi**
Bandas de frecuencia	2.4GHz / 868 MHz / 915 Mhz	2.4GHz	2.4GHz
Tasa de transferencia	250kbps(2.4GHz) / 40kbps(915MHz) / 20kbps(868MHz)	1Mbps	11 Mbps (hasta 54Mbps)
Número de canales	16(2.4GHz) / 10(915MHz) / 1 (868MHz)	79	11 – 14
Tipo de datos	Digital / texto	Digital / Audio	Digital
Rango de nodos internos	10m – 100m	10m – 100m	100m
Número de dispositivos	255 (subred) – 65535 (toda la red)	8	32
Requisitos de alimentación	Años de batería	Días de batería	Horas de batería
Introducción en el mercado	Baja	Alta	Alta
Arquitecturas	Estrella, árbol, punto a punto y malla	Estrella	Estrella
Aplicaciones tipo	Monitorización / control de bajo coste	Ordenadores / teléfono	Internet en un edificio
Consumo de potencia	30 mA transmitiendo / 3mA en reposo	40mA transmitiendo / 0.2mA en reposo	400mA transmitiendo / 20mA en reposo
Precio	Bajo	Medio	Alto
Complejidad	baja	Alta / Media	Baja

La *ZigBee Alliance* define su misión como una asociación de compañías que trabajan juntas para garantizar que la fabricación de productos fiables, baratos, de bajo consumo e inalámbricos orientados a la monitorización y al control basados en un estándar abierto global. Así, la *ZigBee Alliance* pone mucho énfasis en la orientación al control y monitorización, características en las que destaca *ZigBee* en relación al precio de los dispositivos.

2.3.2. Frecuencia de operación, DSSS y mecanismos de acceso

Para minimizar las interferencias en redes suficientemente amplias tanto en extensión como en número de dispositivos, ZigBee utiliza la DSSS (Direct-Sequence Spread Spectrum), a diferencia de Buetooth, que utiliza Frequency-Hopping Spread Spectrum. Básicamente este proceso implica los siguientes elementos:

- DSSS modula en fase una onda senoidal pseudo-aleatoriamente con una cadena continua de códigos de pseudoruido (chips), cada uno de los cuales tiene una duración mucho menor que un bit. Así, cada bit de información se modula siguiendo una frecuencia de chips mucho más rápida. Esto implica que la frecuencia de los chips es mucho mayor que la frecuencia de los bits.
- DSSS emplea una estructura de señal en la cual la secuencia de chips producida por el emisor es conocida a priori por el receptor. Así, el receptor emplea esa misma secuencia para reconstruir la señal original. Para ello las secuencias pseudo-aleatorias en emisor y receptor deben estar sincronizadas.

2.3.3. Pila *ZigBee* y protocolos asociados

La figura 2.3 muestra como el estándar ZigBee se construye a partir de las especificaciones descritas en la norma IEEE 802.15.4.

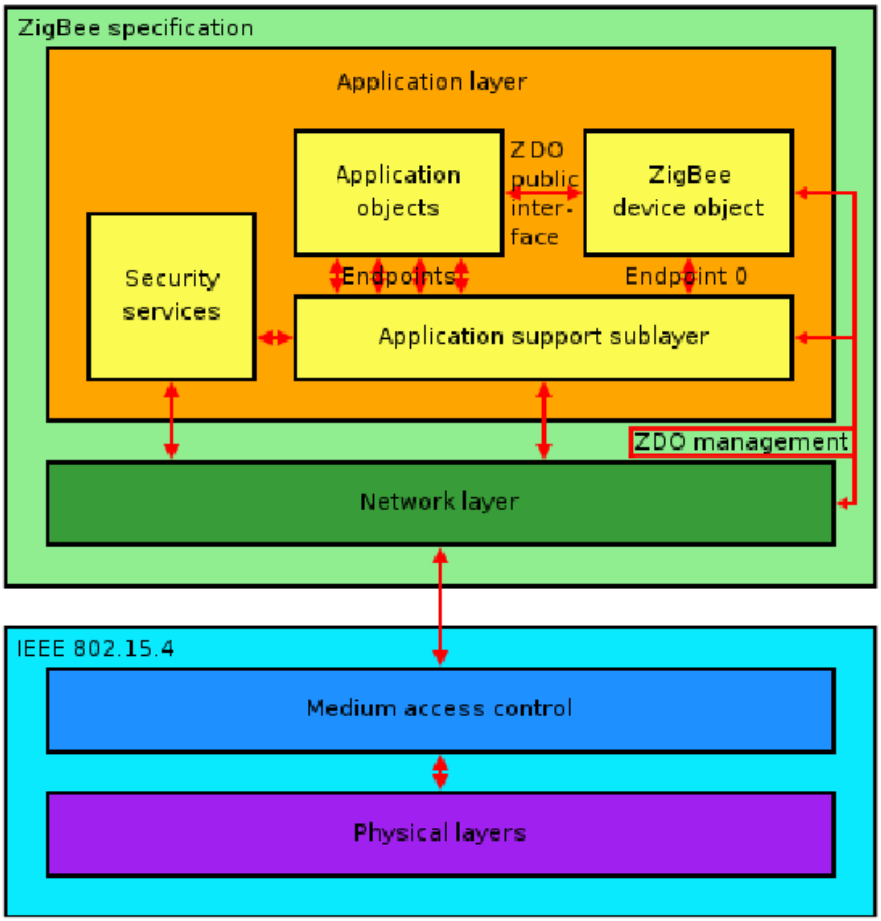

Figura 2.3: Protocolos pila *ZigBee*

La especificación ZigBee se compone de dos capas: la capa de red (network layer) y la capa de aplicación (aplication layer). Pasemos a revisar su estructura:

La capa de red: tiene como principales funciones la de permitir el correcto uso de la capa MAC (previamente definida en la especificación IEEE 802.15.4) y así facilitar una interfaz que pueda ser usada por la siguiente capa (la de aplicación). Ofrece las funcionalidades típicas de una capa de red, incluyendo el enrutado. El protocolo utilizado para el enrutado es el AODV. Este protocolo, para encontrar el dispositivo de destino, emite una petición de enrutado a todos sus vecinos, los cuales, a su vez, repiten la petición de enrutado a sus vecinos, y así sucesivamente hasta que se alcanza el destino. Cuando se alcanza el nodo destino, éste envía la ruta respuesta usando una transmisión unicast (hacia el nodo origen de la petición) usando el camino más corto. Cuando se alcanza el nodo origen de la petición entonces éste actualiza su tabla de enrutado.

La capa de aplicación: esta es la capa de más alto nivel de la especificación, y aquella que sirve de interfaz con el usuario final. Contiene el ZDO (ZigBee Device Object) que se encarga de definir el rol de un dispositivo o bien como coordinador o como nodo final, pero también de detectar nuevos dispositivos en la red y de identificar los servicios que estos ofrecen. Puede llegar a establecer enlaces seguros con dispositivos externos y responder a peticiones en emparejado.La APS (Application Support Sublayer) actúa, a su

vez, como un puente entre la capa de red (network layer) y los demás componentes de la capa de aplicación. Mantiene tablas de enlaces actualizadas que pueden servir para encontrar al dispositivo adecuado en función del servicio que se desee utilizar.

Los Application Objects (AO): pueden consistir en objetos en comunicación que cooperan para llevar a cabo ciertas tareas. ZigBee distribuye estas tareas entre diferentes dispositivos, cada uno ofreciendo un servicio para completar una función. Todos estos objetos, que residen en el mismo nodo, se comunican utilizando los servicios facilitados por APS y ZDO. De hecho, en un único dispositivo pueden existir hasta 240 AOs, con números 1-240. Hay dos servicios disponibles para AOs: el Key-Value Pair (KVP) describe y modifica identificadores de servicios disponibles, y el message service, que permite la transmisión de mensajes si el overhead que implica el KVP. La asignación de direcciones se realiza de forma que cada nodo consiste en un radio emisor-receptor que cumpla con la norma 802.15.4 y una descripción o más de dispositivos, en forma de colecciones de atributos. El emisor-receptor es la base para la asignación de direcciones, y los dispositivos DENTRO de cada nodo se especifican con un identificador final (endpoint identifier) en el rango de 1 a 240.

2.3.4. Tipos de dispositivos y topologías de red

La figura 2.4 muestra la forma en la que se pueden disponer los nodos ZigBee, y los tipos de nodos disponibles.

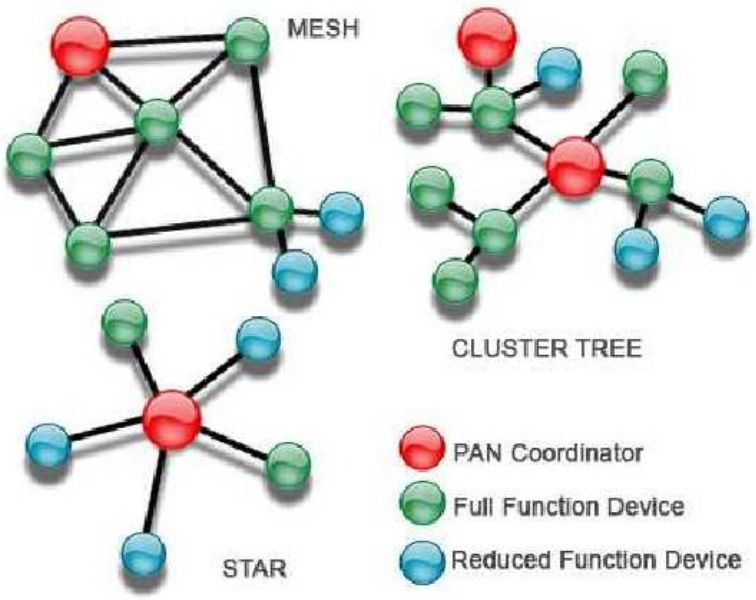

Figura 2.4: Topologías red *ZigBee*

Las topologías permitidas incluyen malla (mesh), estrella (star) y cluster. Para ello es necesario que exista un SOLO coordinador por (sub)red de hasta 254 nodos (255 incluyendo al coordinador). El coordinador, en el paradigma maestro-esclavo, actúa de maestro en cada (subred); mantiene información sobre la (sub)red y determina la mejor ruta para conectar dos nodos. Para redes mayores, se pueden conectar sub redes hasta alcanzar 255 subredes, cada una con su coordinador, permitiendo de esta forma la interconexión de 65535 dispositivos. Dentro de cada (sub)red puede haber routers/repeaters, y dispositivos finales (reduced func-

tion devices). Los nodos con capacidad router/repeater también son dispositivos finales (Full function devices). Todos estos nodos son esclavos.

2.3.5. Algunas aplicaciones prácticas

Algunas de las potenciales aplicaciones de ZigBee en el campo de la domótica y de la automatización de edificios son:

- Integración y centralización de la gestión de la iluminación, calefacción y refrigeración, y seguridad de la vivienda. Control automatizado de múltiples subsistemas con el objeto de mejorar su mantenimiento y seguridad.
- Reducción de la factura energética doméstica a través de una gestión mejorada del aire acondicionado y calefacción.
- Adaptar el entorno doméstico mediante la reconfiguración del sistema de iluminación para generar entornos modulables.
- Extender la infraestructura automatizada de una forma rápida.
- Implementar redes inalámbricas de vigilancia mejorando la protección de la vivienda.

Las potenciales contribuciones de ZigBee en el campo de la salud se sitúan en las siguientes áreas:

- Supervisión y monitorización de ancianos: dispositivos orientados a que la gente mayor disponga de autonomía y a la vez exista un control automático de su estado de salud.
- Control de enfermedades crónicas: para aquellas personas que sufren de algún tipo de enfermedad que requiera una monitorización constante de sus parámetros vitales, tales como la frecuencia cardíaca, la temperatura, la glucosa en sangre, el peso, la saturación...
- Deporte y fitness: monitorización de parámetros tales como las pulsaciones o la velocidad, en dispositivos que apoyen al entrenamiento, tales como pulsómetros o velocímetros.

Capítulo 3

Hardware para la implementación de la comunicación bajo *ZigBee*

3.1. Consideraciones generales

Entre otras opciones, el fabricante elegido para la implementación de la funcionalidad inalámbrica ha sido **Texas Instruments**. La razón de esta elección ha sido la arquitectura del proyecto *weelrobot*[1] *Weelrobot* dispondrá de un dsPIC32 que implementará la comunicación inalámbrica empleando un *SoC* CC2530 comunicado vía serie (empleando los pines 16 y 17) con el dsPIC32, siguiendo el diseño mostrado, en la sección 8.2, en la figura 8.2. Esta es la arquitectura preferida cuando se integra el *SoC* de comunicación con el de control (en este caso un dsPIC) en la misma placa.

Sin embargo, a efectos de desarrollo de software se emplean las herramientas suministradas por los kits de desarrollo de, en este caso, Texas Instruments. Se ha empleado el CCC2530 configurado como una unidad autónoma CC2530EM insertado en una placa SmartRF05EB, para la que ya hay una serie de librerías y que dispone de un puerto USB para la programación del CC2530 en el CC2530EM. También dispone un puerto RS232 que conectado a un conversor Serie/USB se puede conectar a un PC, robot o dispositivo que pueda reconocer un USB Serie y gestionar la comunicación bidireccional. La SmartRF05EB también dispone de una pantalla LCD de tres líneas y de varios LEDs.

Cuando se quiera adaptar el código programado para uso en la placa SmartRF05EB a un *SoC* CC2530 que forme parte de una placa como por ejemplo la del *weelrobot* simplemente se deberán deshabilitar en los correspondientes ficheros de configuración los periféricos no presentes, y dejar habilitados aquellos que siguen presentes. En este caso particular el único presente será el puerto serie. El resto de funciones que hagan referencia a periféricos inexistentes, al estar deshabilitados por software, no se ejecutarán.

[1] Cuya fecha de entrega era Septiembre del 2011 y por razones totalmente ajenas al control del autor, pospuesta *sine die*.

3.2. Descripción del *SoC* CC2530 y del CC2531

La figura 3.1 muestra la estructura del *SoC CC2530*, módulo *ZigBee* de Texas Instruments.

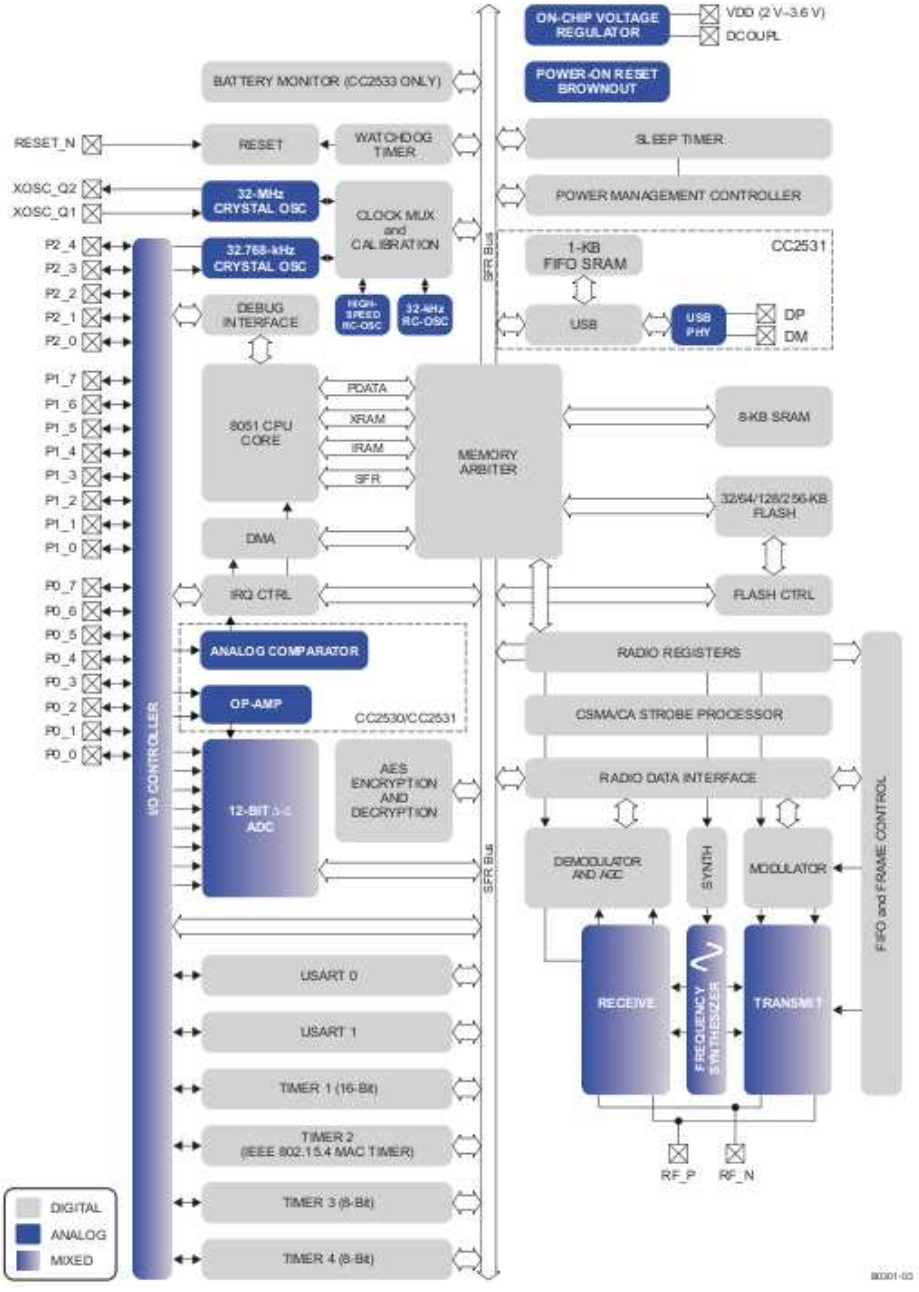

Figura 3.1: Esquema del CC253x

Se describen a continuación las principales características del CC2350.Para una descripción más exhaustiva se puede consultar la referencia [6].

3.2.1. CPU y memoria

El CC2530 posee un **procesador 8051** monociclo. Dispone de tres buses de acceso a memoria diferentes (SFR, DATA y CODE/XDATA) con acceso monociclo a SFR, DATA y la SRAM principal. También incluye un interfaz para debugging y una unidad de interrupciones extendida de 18 bits.

El **controlador de interrupciones** controla 18 posibles fuentes de interrupción, divididas en 6 grupos, cada uno asociado a una de las cuatro prioridades de interrupción posibles. Cualquier peticiónn de interrupcciónnse atiende aunque incluso si el dispositivo se encuentra en modo de reposo (ïdle") revirtiendo a un estado activo. Algunas interrupciones también puede "despertar.[el] dispositivo del modo durmiente ("sleep mode"). De hecho, cuando el dispositivo se encuentra en modo durmiente, éste se encuentra en uno de los tres posibles modos de bajo consumo: PM1, PM2 y PM3.

El **controlador de memoria** (memory arbiter) es una de las partes más importantes del sistema, puesto que conecta la CPU y controlador DMA con la memoria física y con los periféricos a través del bus SFR. el **controlador de memoria** tiene cuatro puntos de acceso a memoria, accesos que pueden apuntar a una de las tres memorias disponibles: SRAM, memoria flash y registros XREG/SFR. Se encarga de arbitrar y secuenciar peticiones de acceso simultáneas a la misma memoria.

La **4/6/8-KB SRAM** apunta al espacio de memoria DATA y a ciertas secciones de XDATA. La SRAM es una memoria de extremadamente bajo consumo con contenidos persistentes en todos los modos de alimentación. Esta característica es especialmente relevante en aplicaciones de bajo consumo.

Los **bloques 32/64/96/128/256 KB de memoria flash** ofrecen el almacenamiento no volátil del programa para el dispositivo, y mapea a los espacios de memoria CODE y XDATA. Además de contener código de programa y constantes, la memoria no volátil permite a la aplicación el grabar datos que se deben preservar, de forma que estén disponibles cuando se reinicie el dispositivo.

3.2.2. Relojes y gestión energética

El núcleo digital y los periféricos se alimentan via un regulador de voltaje de 1.8V. Además, el CC2530/CC2531 posee una función de control de potencia que permite el uso de diferentes modos de bajo consumo (PM1, PM2 y PM3) para aplicaciones de bajo consumo y de larga duración de las baterías.

3.2.3. Periféricos

Los microcontroladores CC253x/CC2540 ofrecen (dependiendo del modelo) diferentes periféricos que permiten la programación de aplicaciones avanzadas.

El **interfaz de debugging** implementa una interfaz propietaria de dos cables que se emplea para el debugging en el propio circuito. A través de esta interfaz es posible completar el borrado de la memoria flash, controlar el activado de los osciladores, iniciar y detener la ejecución del programa, ejecutar las instrucciones en el 8051 y definir breakpoints.

El dispositivo contiene **memoria flash** para alojar el programa. La memoria flash es programable desde el interfaz de debugging. El **controlador flash** controla la escritura y la lectura en la memoria flash.

El **controlador I/O** controla los pines I/O de propósito general. La CPU puede configurar si los periféricos controlan ciertos pines o, por el contrario, se encuentran bajo control del software, y en ese caso, si cada pin se configura como Input o Output. Las interrupciones a la CPU se pueden activar en cada pin individualmente. Cada periférico conectado a los pines I/O tiene la opción de elegir entre dos localizaciones para los mismos, garantizando así la flexibilidad al cambiar de aplicación.

El **controlador DMA** de cinco canales accede a la memoria usando el espacio XDATA y, por tanto, tiene acceso a todas las memorias físicas. Cada canal esta configurado con descriptores DMA en cualquier lugar de la memoria. Muchos de los periféricos obtienen gran eficiencia de operación usando el controlador DMA para la transferencia de datos entre direcciones SFR o XRED y flash/SRAM.

El **Timer 1** es un timer de 16-nits con funcionalidad timer/counter/PWM. Tiene disponible un preescalado de 16 bits y cinco contador/captura canales que se pueden programar de forma individual con un valor de comparación de 16 bits. Cada uno de los canales puede ser utilizado como un PWM Output o para capturar los flancos de las señales de entrada

El **Timer 2** (el timer MAC) está diseñado específicamente para dar soporte al IEEE 802.15.4 MAC o cualquier otro protocolo que utilice slots de tiempo reservados. El timer dispone de un timer configurable y un contador de 24 bits de overflow que se puede emplear para llevar la cuenta del número de periodos que han transcurrido. Un registro de 40 bits también se emplea para registrar el momento exacto en el cual un delimitador de inicio de trama se recibe o transmite o el momento exacto en el que esta transmisión termina, así como dos registros de comparación de salida de 16 bits y dos registros de 24 bits de comparación de overflow que pueden enviar comandos al módulo de radio.

Los **Timers 3 y 4** son timers de 8 bits con una triple funcionalidad timer/counter/PWM. Disponen de un preescalado programable, un periodo de 8 bits y un canal counter programable con un valor de comparación de 8 bits. Cada uno de los canales se puede usar como salida PWM.

El **Sleep Timer** es un timer de ultra bajo consumo que cuenta con un cristal oscilador de 32-KHz. El Sleep Timer funciona continuamente en todos los modos de operación excepto en el PM3. Se usa para aplicaciones como contador en tiempo real o para despertar del estado durmiente al sistema para salir de los modos PM1 y PM2.

El **ADC** soporta de 7 bits (ancho de banda 30kHz) hasta 12 bits (con ancho de banda 4kHz). DC y conversión de audui son posibles con hasta ocho canales de entrada (Puerto 0) de resolución. Los inputs se pueden seleccionar como single-ended o como diferenciales. El voltaje de referencia puede ser interno o externo. El ADC también dispone de un canal para un sensor de temperatura. El ADC puede automatizar el proceso de muestreo periódico o conversión sobre una secuencia de canales.

El **generador de número aleatorio** utiliza un registro especial para generar números pseudoaleatorios que pueden ser leídos directamente por la CPU.

El **coprocesador AES** permite al usuario cifrar y descifrar datos utilizando el algoritmo AES con claves de 128 bits. El núcleo es capaz de completar las operaciones de seguridad requeridas

por el IEEE 802.15.4 MAC, la capa de red de *ZigBee* y la capa de aplicación.

Los **USART 0 y USART 1** se pueden configurar como SPI esclavo/maestro o UART. Ofrecen un doble buffering tanto en RX como en TX, y están diseñados para aplicaciones full-duplex de alta carga. Ambos disponen de su propio reloj de alta precisión, liberando los Timers ordinarios para otras aplicaciones.

El **controlador USB 2.0** (en el CC2531) opera en full-speed con un ancho de banda de 12Mbps. Dispone de cinco endpoints bidireccionales además del endpoint 0 (usado para control). Estos endpoints soportan las modalidades de transmisión de USB, a saber:

- Bulk
- Interrupt
- Isochronus

El **amplificador operacional** (CC2530 y CC2531) se emplea en el ADC.

El **comparador analógico de ultrabajo consumo** (CC2530 y CC2531) permite a las aplicaciones "despertar"de los estados PM2 y PM3 en función de una señal analógica

3.2.4. Radio

Los dispositivos de la familia CC253x proporcionan un emisor/receptor de radio que cumple con la norma IEEE 802.15.4. El núcleo de radiofrecuencia controla el módulo analógico de radio. Además, proporciona la conexión entre el microcontrolador y la radio, lo que permite ejecutar comandos y leer datos del estado.

3.3. Descripción del CC2350EM

Para el desarrollo de la aplicación puente objeto del presente documento se ha empleado el CC2530 en su forma implementada en la placa CC2530EM, que se puede ver en la figura 3.2.

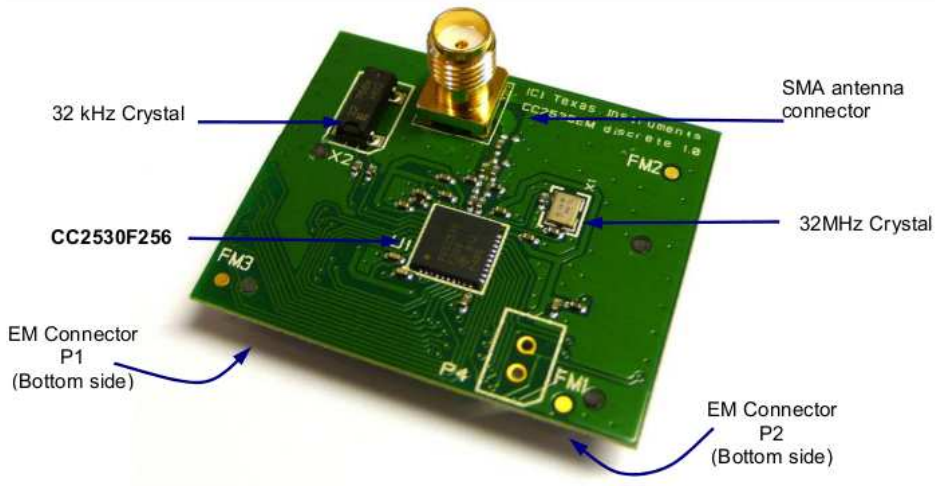

Figura 3.2: CC2530

Esta implementación permite su inserción en las placas *SmartRF05 Evaluation Board* y *SmartRF05 Battery Board*.

el CC2530 es un módulo completo de radiofrecuencia basado en uno de los diseños recomendados por Tezas Instruments. El módulo está equipado con un cristal de 32 MHz, otro cristal de 32.768kHz, componentes pasivos externos que actúan de filtro sobre la antena, y las conexiones para la antena y la placa en la que va insertado.

3.4. Descripción del SmartRF05 Evaluation Board

Para el desarrollo de la aplicación puente se ha trabajado sobre la placa madre *SmartRF05 Evaluation Board*, que comprende una serie de dispositivos y periféricos que permiten la interacción entre el **CC2530** en su forma **CC2530EM**[2] y una serie de dispositivos que se pueden controlar usando la API que facilita Texas Instruments y que se describirá en el capítulo 4.

La versión de la *SmartRF05 Evaluation Board* empleada es la revisión 1.8.1. El aspecto de la placa se puede ver en la figura 3.3.

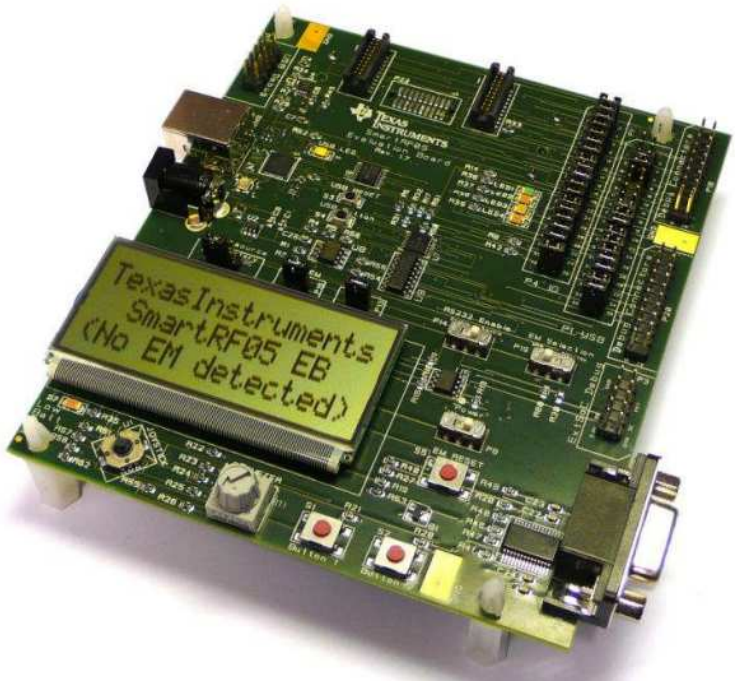

Figura 3.3: SmartRF05 Evaluation Board

La figura 3.4 muestra la *SmartRF05 Evaluation Board* con el módulo *CC2530EM* insertado, junto con la antena.

[2]También se puede emplear para implantar módulos de la familia CC diferentes al CC2530

Figura 3.4: SmartRF05 Evaluation Board con CC2530EM

El principal componente de la placa es el controlador USB. Se comunica con el PC via USB y traduce las peticiones de diversas herramientas informáticas que pueden correr sobre el PC. El controlador USB se comunica con el CC2530EM via SPI, UART y/o la interfaz para debugging. Es importarte resaltar que no todos los periféricos en la placa son accesibles desde el controlador USB. De hecho, tiene accso al UART RS232, al LCD, el LED D6, el joystick y un botón (el botón reset del USB). No dispone de acceso a la memoria flash serie.

El módulo EM (*CC2530*) puede disponer de acceso a todos los periféricos de la placa. Dispone de acceso pleno al LCD, flash serie, cuatro LEDs, dos botones, joystick y UART RS232.

Puesto que a muchos de los periféricos se puede acceder o bien desde el controlador USB o bien desde el CC2530, algunos pines de I/O pueden ser potencialmente manejados por ambos. El firmware del controlador USB gestiona esto activando los pines I/O compartidos en triestado (alta impedancia), y así eliminando potenciales conflictos. La arquitectura interna se puede ver en la figura 3.5.

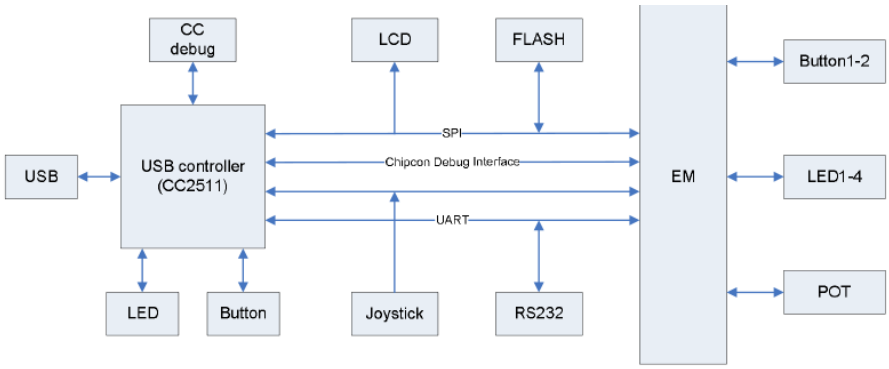

Figura 3.5: Arquitectura de la *SmartRF05 Evaluation Board*

La figura 3.6 muestra la *SmartRF Evaluation Board* versión 1.8 con la descripción de sus principales componentes.

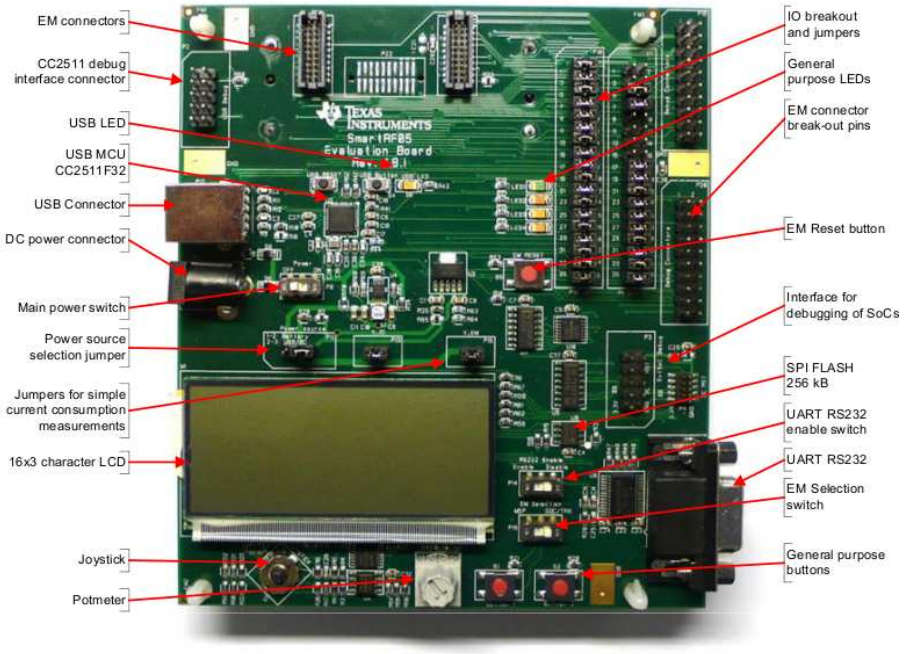

Figura 3.6: SmartRF05 Evaluation Board con CC2530EM y explicación de sus componentes

Para una información más detallada referirse al documento de Texas Instruments [12]. Para una discusión genérica de la programación de empotrados en C y C++ se puede consultar la referencia [17].

Capítulo 4

Software en la implementación bajo *ZigBee* para los equipos de Texas Instruments

4.1. Descripción del entorno de desarrollo: IAR

Para el desarrollo del software necesario para la ejecución del proyecto se ha trabajado con el compilador IAR para el 8051. Sobre él se ha descargado la librería de Texas Instruments *Z-stack*, programada en C, que contiene los .h y los .c necesarios para la programación del *CC2530EM* sobre la placa *SmartRF05 Evaluation Board*, puesto que incorpora los drivers necesarios para, desde una abstracción de software, acceder a los periféricos de la placa. Esto se verá en las secciones 4.3 y 4.4. Una discusión más exhaustiva de las opciones de compilación del Z-stack se pueden encontrar en la referencia [9].

El compilador IAR funciona bajo entorno Windows y se descarga de la página de descarga www.iar.com. Se puede solicitar una licencia de prueba válida durante 30 días. Tras la instalación, el IDE permite trabajar en C. Existe un fichero de configuración del workspace donde se le indica al proyecto dónde buscar los .h y los.c así como las preferencias de compilación y la predefinición de símbolos. El aspecto del IDE se puede ver en la figura 4.1.

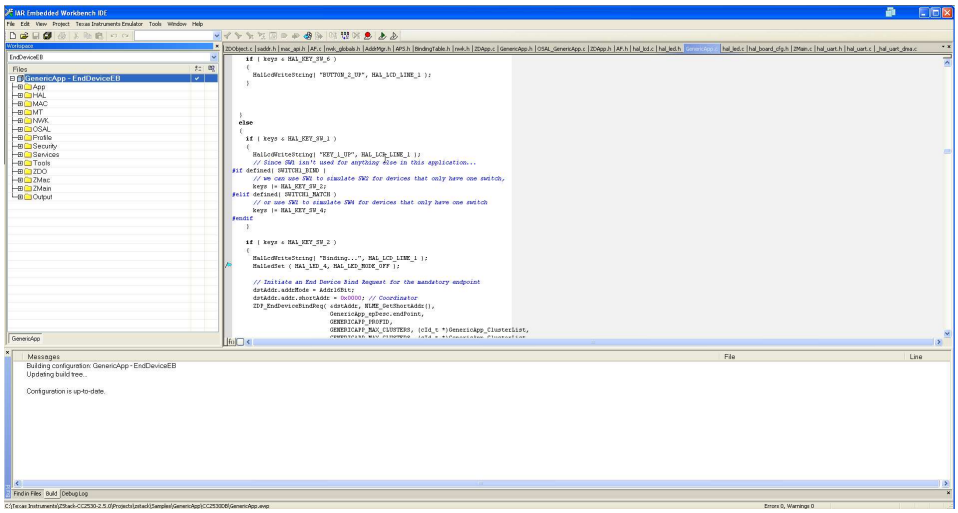

Figura 4.1: IDE IAR

En la parte izquierda se puede ver el navegador del proyecto, donde aparecen referenciados todos los ficheros que forman parte del proyecto (tanto los .h como los .c), organizados en carpetas. En la ventana derecha se ve el espacio de programación así como las diferentes pestañas de los documentos abiertos. Finalmente en la parte inferior del IDE se puede ver el espacio para mensajes generados durante la compilación.

El IAR permite via USB la programación del *CC2530EM* cuando éste se encuentra insertado en la *SmartRF05 Evaluation Board*.

Para más información se puede consultar el documento [10].

4.2. Estructura de un proyecto de software en IAR

Cualquier proyecto de software en el IAR para programar *ZigBee* en los CC2530 y CC2531 incluye una serie de librerías que conjuntamente se denominan *Z-Stack*. Dependiendo de la natuaraleza del proyecto, se referencian más o menos elementos. En cualquier caso cabe destacar los siguientes elementos en cualquier proyecto:

ZMain: incluye, entre otros, un fichero, el **ZMain.c**. Este fichero realiza las tareas previas a la inicialización del sistema operativo ofrecido en el *Z-Stack* y que se emplea como estructura de programación posterior. Entre otras tareas inicializa periféricos y cuando ha acabado lanza el sistema operativo, que desde el punto de vista del proyecto es el último proceso (del que no habrá retorno) que se invoca desde el main de **ZMain.c**.

App: Es la carpeta que contiene la aplicación propiamente dicha, puesto que el resto de ficheros incluidos en el proyecto son librerías o ficheros de configuración. Esta carpeta incluye, normalmente tres ficheros. Supongamos que el nombre elegido para la aplicación es "GenericApp". En ese caso los ficheros serían:

- **GenericApp.h** que comprendería los prototipos de las funciones propias de la aplicación.

- **GenericApp.c** que comprendería la implementación de las funciones propias de la aplicación. Incluye la función principal de gestión de eventos, que permite, por ejemplo, procesar un mensaje *ZigBee* entrante, o, si fuera necesario, el evento derivado de la pulsación de un botón o del movimiento del joystick.
- **OSAL_GenericApp.c** que incluye una función que se debe implementar en este fichero (*osalInitTasks(void)*) y un array del tipo *pTaskEventHandlerFn* (TaskArr[]). Para una determinada lista de tareas a asignar al sistema operativo se realiza una inicialización (osalInitTasks) y posteriormente se describe la rutina asociada a cada inicialización *en el mismo orden en que se ha inicializado*. Esta rutina, de ser propia de la aplicación se incluye en GenericApp.c.

HAL (Hardware Abstraction Layer): puesto que se asume que el chipset de la placa donde va insertado el *CC2530EM* corresponde la configuración de hardware *SmartRF05 Evaluation Board* o *SmartRF05 Battery Board*, Texas Instruments ya suministra una abstracción de software que permite acceder a periféricos como el LCD, los LED, o recibir eventos al pulsar botones o el joystick, o al recibir información via UART. De entre los .c y .h cabe destacar el fichero **hal_board_cfg.h**, que contiene las definiciones que permiten realizar las funciones asignadas a ciertos periféricos en el caso de existir. Por ejemplo

Listing 4.1: definiciones

```
1 #ifndef HAL_LCD
2 #define HAL_LCD TRUE
3 #endif
```

describe la existencia de un visor LCD *de acuerdo a la configuración de hardware de las placas SmartRF05*. Si en vez de TRUE fuera FALSE entonces la programación interna de los drivers del LCD haría que las órdenes asociadas al LCD se ignoraran. Esto es especialmente últil puesto que respetando la conexión a los puertos UART (pines 16 y 17 del *SoC* CC2530) todo el código definido en las placas de desarrollo es portable a cualquier equipo que implemente este *SoC*, teniendo la precaución de definir como FALSE aquellos periféricos que no estarían presentes en la placa final.

OSAL (Operating System Abstraction Layer): contiene las librerías asociadas a la implementación del sistema operativo ofrecido por la librería, que permite una programación basada en eventos que se gestionan a partir de rutinas específicas. El sistema operativo detecta las interrupciones de hardware o software y las convierte en mensajes o eventos, añadiendo así una capa extra de abstracción que facilita la confección de rutinas de tratamiento de eventos.

Tools: esta carpeta contiene, a diferencia de las anteriores, una serie de ficheros de configuración *xcl* y *cfg*. El IDE permite compilar el código con uno de los tres roles de los dispositivos *ZigBee*, a saber, End Device, Router y Coordinador. Esta carpeta contiene cinco ficheros:

- **f8w2530.xcl**
- **f8wconfig.cfg**: en este fichero se determinan cosas como el PANID de la red a generar (si se es Coordinator) o a la que unirse (si se es Router o End Device)
- **f8wcoord.cfg**: se incluye sólo en el caso de que se compile el proyecto como *coordinator*.

- **f8wenddev.cfg**: se incluye sólo en el caso de que se compile el proyecto como *end device*.
- **f8wrouter.cfg**: se incluye sólo en el caso de que se compile el proyecto como *router*.

Profile : esta carpeta incluye los ficheros **AF.c** y **AF.h**, que exponen la funcionalidad denla pila *ZigBee* que es realmente accesible al programador.

Otras carpetas: NWK,**Security**,**ZDO**... que contienen librerías que emplea directamente el programador o que sirven de soporte a las que emplea.

4.3. Z-Stack de Texas instruments para CC2530 y CC2531: Application Framework

Cuando se instala el paquete de librerías que componen el Z-Stack, en realidad se instalan .c y .h que corresponden:

- A la capa de acceso del programador a la funcionalidad *ZigBee*, que como se ha visto en la sección 4.2 corresponden a aquellas que el programador puede invocar directamente (en **AF.c** y **AF.h**)
- A capas intermedias del protocolo *ZigBee* que necesita el *Application Framework* para funcionar correctamente
- Otras funcionalidades necesarias para *ZigBee* que, sin embargo, serían necesarias o bien en proyectos sin funcionalidades inalámbricas (por ejemplo, OSAL o HAL), o bien otras que serían necesarias en proyectos con funcionalidades inalámbricas que cumplieran con la especificación 802.15.4 pero que no fueran necesariamente *ZigBee*. De hecho, Texas Instruments incluye protocolos para comunicaciones inalámbricas que cumplen con la IEEE 802.15.4. pero que son diferentes (y más sencillos) que *ZigBee*.

Por lo anterior, lo que realmente es el procedimiento, dada toda la funcionalidad ofrecida por las herramientas de Texas Instruments, que es necesario seguir para emplear *ZigBee* bajo el *CC2530*. Para más información sobre la sintaxis general se puede consultar el documento [14]. En la presene sección se explicará el mecanismo de envío de paquetes y la función de la API necesaria.

Esta tesina se va a centrar en los mecanismos de direccionamiento, envio y recepción de mensajes a direcciones conocidas. Existe una alternativa, empleando la librería ZDO (*ZigBee Device Objects*) que permite construir, siguiendo un procedimiento en ejecución, una tabla de emparejado para diferentes dispositivos, de forma que al enviar se emplean las direcciones que se encuentran en la tabla.

En cuanto al direccionamiento, una red *ZigBee* tiene dispositivos de tres tipos:

- Coordinator
- Routers
- End Device

Toda red dispone de un PANID, que se fija o bien aleatoriamente o bien definiéndolo en el fichero f8wcoord.cfg. Cada red sólo tiene un **coordinator** cuya dirección, de 16 bits, es 0x0000. El resto de las direcciones a **routers** y a **end devices** se asignan en función de su pertenencia a la red. Cada dirección puede contener hasta 240 *end points* y cada end point puede tener *clusters* de comandos estandarizados, en caso de utilizar un determinado perfil, asociado al dispositivo. El problema de la comunicación entre robots se ha resuelto con un único *end point* (por convenio se ha utilizado el número 10) y sin clusters.

Para enviar un mensaje *ZigBee* se utiliza la función **AF_DataRequest()**. Un ejemplo de utilización es el siguiente:

Listing 4.2: AF_DatarRequest

```
1
2 if ( (GenericApp_NwkState == DEV_END_DEVICE) )
3   {
4     HalLcdWriteString( "Mensaje Enviado", HAL_LCD_LINE_2 );
5     if ( AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc,
6                          GENERICAPP_CLUSTERID,
7                          (byte)osal_strlen( theMessageData ) + 1,
8                          (byte *)&theMessageData,
9                          &GenericApp_TransID,
10                         AF_DISCV_ROUTE, AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
11    {
12      // Successfully requested to be sent.
13    }
14    else
15    {
16      // Error occurred in request to send.
17    }
18 }
```

cuyo prototipo es el siguiente y se encuentra en AF.c:

Listing 4.3: Prototipo de AF_DataRequest

```
1 /*********************************************************************
2  * @fn      AF_DataRequest
3  *
4  * @brief   Common functionality for invoking APSDE_DataReq() for both
5  *          SendMulti and MSG-Send.
6  *
7  * input parameters
8  *
9  * @param  *dstAddr - Full ZB destination address: Nwk Addr + End Point.
10 * @param  *srcEP - Origination (i.e. respond to or ack to) End Point Descr.
11 * @param   cID - A valid cluster ID as specified by the Profile.
12 * @param   len - Number of bytes of data pointed to by next param.
13 * @param  *buf - A pointer to the data bytes to send.
14 * @param  *transID - A pointer to a byte which can be modified and which will
15 *                    be used as the transaction sequence number of the msg.
16 * @param   options - Valid bit mask of Tx options.
17 * @param   radius - Normally set to AF_DEFAULT_RADIUS.
18 *
19 * output parameters
20 *
21 * @param  *transID - Incremented by one if the return value is success.
22 *
23 * @return  afStatus_t - See previous definition of afStatus_... types.
```

```
24  */
25
26 afStatus_t AF_DataRequest( afAddrType_t *dstAddr, endPointDesc_t *srcEP,
27                            uint16 cID, uint16 len, uint8 *buf, uint8 *transID,
28                            uint8 options, uint8 radius )
```

El dispositivo de destino se describe con un *struct* afAddType_t:

Listing 4.4: Estructura afAddType_t

```
1  typedef struct
2  {
3    union
4    {
5      uint16      shortAddr;
6      ZLongAddr_t extAddr;
7    } addr;
8    afAddrMode_t addrMode;
9    uint8 endPoint;
10   uint16 panId;  // used for the INTER_PAN feature
11 } afAddrType_t;
```

Esta definición se encuentra en AF.h, y su empleo permite una comunicación *unicast*, es decir, dada la dirección de destino (network address + end point) se envía un paquete de datos que se encapsula en un array cuya longitud y dirección se incluyen en los argumentos de la función. *ZigBee* se encarga de que los datos se envíen correctamente.

Existen otras alternativas de direccionamiento, como:

Broadcast : envío a todos los dispositivos de la red.

Indirect : usando la **binding table**.

Group addressing : enviando a un grupo de dispositivos

4.4. Otras librerías importantes

4.4.1. HAL

La HAL es la capa de abstracción que ofrece los servicios del hardware (referidos a el chipset *SmartRF05EB* y *SmartRF05BB*) sin necesidad de exponer demasiados detalles de bajo nivel sobre el hardware, de forma que el programador pueda centrar su atención en la aplicación.

La mayoría del material incluído en la libreria son los drivers que controlan los LEDs, el LCD,ADC, teclas (joystick y botones), los timers y el UART. A estos servicios se accede vía la abstracción que ofrecen esta API, evitando al usuario el tener que programar el *SoC* a nivel de GPI/O[1].

La librería se estructura en tres directorios:

- Common

[1] General Purpose Input Output

- Include
- Target

que deben ser incuídos en la ruta de búsqueda del compilador.

Cada directorio contiene una serie de ficheros que ofrecen servicios:

Common : contiene, entre otros, los siguientes elementos:

- **hal_assert.c**, empleado para la verificación de datos
- **hal_driver.c**, que contiene funciones de inicialización como *Hal_Init()* (empleada por osalTaskAdd() para registrar drivers como una tarea en el OSAL), *HalDriverInit()*, *Hal_ProcessEvent()* y *Hal_ProcessPoll()*.

Include : contiene los .h de los drivers y de los ficheros relacionados con la HAL

- hal_adc.h
- hal_key.h
- hal_lcd.h
- ...

Target : contiene los ficheros específicos para el chipset. Incluye drivers, ficheros de configuración y tipos. Incluye

- hal_key.c
- hal_lcd.c
- hal_led.c
- hal_board_cfg.h
- hal_target.h
- hal_types.h
- ...

Para más información consultar los documentos [8] y [7].

4.4.2. OSAL

La capa de abstracción del sistema operativo (OSAL) se emplea para desacoplar los componentes del Z-stack de los detalles de más bajo nivel. Ofrece funcionalidades en las siguientes áreas.

- Registro, inicialización y activación de tareas
- Asignación de memoria
- Intercambio de mensajes entre tareas
- Sincronización de tareas

- Gestión de las interrupciones
- Timers

Los anteriores servicios se sustancian en una serie de APIs integradas en OSAL:

API de gestión de mensajes : ofrece un mecanismo para intercambiar mensajes entre tareas o elementos de proceso con diferentes entornos, como, por ejemplo, rutinas de servicio de interrupcion o tareas invocadas dentro de un bucle. Las funcionas en esta API permiten asignar y liberar buffers para mensajes, enviar mensajes que encapsulan comandos a otras tareas y recibir mensajes desde otras tareas.

API de sincronización de tareas : esta API faculta a una tarea a esperar a que un determinado evento tenga lugar y, durante la espera, ceder el control a otra tarea mientras se completa la espera. Las funciones en esta API se pueden emplear para habilitar eventos para una determinada tarea y notificarlos a esta tarea una vez tengan lugar

API de gestión de timers : permite a una tarea emplear timers software. Suministra funciones para iniciar y detener timers, los cuales se pueden ajustar en incrementos de 1 ms.

API de gestión de interrupciones : faculta una tarea el acceso a interrupciones externas. Las funciones en la API permiten a las tareas el asociar una rutina específica con cada interrupción, las cuales se pueden encontrar habilitadas o deshabilitadas.

API de gestión de tareas : se emplea para añadir y gestionar tareas vía OSAL.

API de gestión de memoria : representa un método simple de asignación de memoria. Las funciones de esta API permiten la asignación dinámica de memoria.

API de gestión de alimentación : se emplea para suministrar a las tareas y aplicaciones un mecanismo que notifique al OSAL cuando es seguro desconectar el receptor y le hardware externo, y a su vez llevar al procesador al modo durmiente.

API de gestión de la memoria no volátil : el sistema ofrece a las aplicaciones una forma de almacenar información de forma persistente en dispotivos de memoria adecuados. También se emplea por el Z-Stack para el almacenamiento persistente de ciertos datos requeridos por la especificación *ZigBee*.

Para más información y para la descripción de las funciones incluíadas en la API se puede consultar el documento [3]. Para una discusión generalista de los sistemas operativos se puede consultar la referencia [15]. Aunque esta OSAL no es un RTOS, puede resultar interesante la referencia [20].

4.5. Compilación y programación

El IAR permite la compilación del programa y la generación de una serie de ficheros listos para la programación del CC2530 (o de cualquier otro target). Aunque se puede elegir el tipo de fichero, lo más usual es emplear para el output un fichero tipo hex.

Una vez se obtiene el fichero hex lo ideal es utilizar el mismo IAR para realizar la programación del CC2530EM que se encuentra insertado en la placa SmartRF05EB. Para ello se emplea un cable USB, que permite al PC reconocer el dispositivo siempre que éste tenga instalado el correspondiente driver. La forma más sencilla de instalar este driver es instalando el programa **SmartRF Flash Programmer**. En teoría este software permite, dado un fichero hex, programar el CC2530 independientemente del IDE empleado para generar el hex. Sin embargo, la experiencia ha mostrado que el **SmartRF Flash Programmer** puede dar problemas en la programación del CC2530, y estos problemas no se han detectado cuando se ha procedido desde el IAR.

Resumiendo, para programar el CC2530 en su forma CC2530EM insertado en la placa SmartRF05EB se emplea un cable USB, y se programa desde el IAR. Sin embargo, para que el IAR reconozca el dispositivo es necesario instalar el driver, y la mejor forma de hacerlo es instalando el SmartRF Flash Programmer.

Capítulo 5

Estandarización de la trama *ZigBee*

5.1. Justificación de la necesidad de estandarizar la trama

El objeto del presente proyecto es presentar un modelo que permita estandarizar la comunicación entre robots (o incluso otros equipos) empleando una sintaxis simple que se base en dispositivos presentes en los *SoC* y en los empotrados en general.

Una vez se ha seleccionado *ZigBee* como tecnología de comunicación inalámbrica dada la gran flexibilidad en cuanto a topologías, su bajo coste y su bajo consumo, hay que plantearse que tipo de interacción habrá con el equipo que se encuentra al otro lado del *SoC*. Lo lógico es plantearse una interacción via UART, que en este caso sigue las siguientes especificaciones UART:

- 9600 baudios
- palabras de 8 bits
- sin bit de control
- un stop bit
- el stop bit en ALTO

Esta es la configuración estándar de la comunicación serie, y la que se empleará en la programación.

Así, al otro lado del *SoC* se encuentra el microcontrolador o PC que realmente toma decisiones y que o bien recibe información o bien la envía. Si recibe información debe saber de dónde viene y entender los comandos incluídos en la trama, y si la envía la trama debe incluir, además del comando y los datos, la dirección de destino. Cuando el CC2530 la procese, sabrá a dónde enviarla porque en la trama buscará la dirección de destino en ciertas posiciones de la trama, donde el PC o el microcontrolador la habrán puesto.

5.2. Estructura propuesta

La estructura propuesta para la trama es la siguiente:

byte 0 : siempre 0xFF

byte 1 : dirección de origen, LSB

byte 2 : dirección de origen, MSB

byte 3 : end point de origen

byte 4 : dirección de destino, LSB

byte 5 : dirección de destino, MSB

byte 6 : end point de destino

byte 7 : cluster de destino

byte 8 : comando, LSB

byte 9 : comando, MSB

byte 10-30 : datos, de LSB a MSB

byte 31 : '\0'

Si el mensaje es **entrante** en el *SoC* **desde UART**, el CC2530 toma los datos de los bytes 4 y 5 y construye la dirección de destino, y el byte 6 se usa para el end point de destino. Además rellena los bytes 1 y 2 (con la dirección de origen) y el byte 3 (end point de origen). El microcontrolador o PC previo debe, por tanto, haber rellenado los bytes 8 y 9 (de comandos), los bytes del 10 al 30 (de datos) así como los bytes 4 y 5 de destino.

Si el mensaje es **entrante** en el *SoC* **desde el módulo inalámbrico**, el CC2530 se limita a escribir los 32 bytes en UART, para que el microcontrolador o PC conectado vía serie reciba la trama completa y actúe en consecuencia.

Notar que el primer byte y el último byte de la trama son siempre los mismos valores (0xFF y '0') pudiendo este patrón, junto con la longitud fija de 32 octetos ser utilizado para garantizar la integridad de la trama.

5.3. Comandos e interpretación de las diferentes versiones de la trama

Cuadro 5.1: Listado de algunos posibles comandos y la estructura de los argumentos

N(0-31)	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	EXPLICACION
	COMANDO	COMANDO	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS	ARGMTOS		
	LSB	MSB	char	char	char	char	char	char	char	char	char	char	char	char	char	char	char	char	char	char	char	char	char	
COMANDOS	00	00	unsigned char* con el último octeto empleado a ” (0x00)																					MENSAJE DE IDENTIFICACIÓN
	01	00	unsigned char* con el último octeto empleado a ” (0x00)																					CONFIRMACIÓN RECEPCIÓN
	02	00	unsigned char* con el último octeto empleado a ” (0x00)																					CONFIRMACIÓN EJECUCIÓN CORRECTA
	03	00	unsigned char* con el último octeto empleado a ” (0x00)																					CONFIRMACIÓN EJECUCIÓN INCORRECTA
	04	00	X-Hm	X-Dm	X-metros	X-dm	X-cm	X-mm	':'	Y-Hm	Y-Dm	Y-metros	Y-dm	Y-cm	Y-mm	':'	Z-Hm	Z-Dm	Z-metros	Z-dm	Z-cm	Z-mm	”	TRASLACIÓN ABSOLUTA
	05	00	X-Hm	X-Dm	X-metros	X-dm	X-cm	X-mm	':'	Y-Hm	Y-Dm	Y-metros	Y-dm	Y-cm	Y-mm	':'	Z-Hm	Z-Dm	Z-metros	Z-dm	Z-cm	Z-mm	”	TRASLACIÓN RELATIVA
	06	00	X-Hm	X-Dm	X-metros	X-dm	X-cm	X-mm	':'	Y-Hm	Y-Dm	Y-metros	Y-dm	Y-cm	Y-mm	':'	º-C	º-D	º-U	º-d	º-c	º-mil	”	TRAS-ROTA. ABSOLUTA
	07	00	X-Hm	X-Dm	X-metros	X-dm	X-cm	X-mm	':'	Y-Hm	Y-Dm	Y-metros	Y-dm	Y-cm	Y-mm	':'	º-C	º-D	º-U	º-d	º-c	º-mil	”	TRASLACIÓN-ROTACION RELATIVA
	08	00	º-C	º-D	º-U	º-d	º-c	º-mil	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	”	ROTACIÓN ABSOLUTA
	09	00	º-C	º-D	º-U	º-d	º-c	º-mil	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	':'	”	ROTACIÓN RELATIVA
	0A	00	X-Hm	X-Dm	X-metros	X-dm	X-cm	X-mm	':'	Z-Hm	Z-Dm	Z-metros	Z-dm	Z-cm	Z-mm	':'	º-C	º-D	º-U	º-d	º-c	º-mil	”	POSICIONAMIENTO PINZA
	0B	00	unsigned char* con el último octeto empleado a ” (0x00)																					MENSAJE ORDEN COMPLETADA
	0C	00																						
	0D	00																						
	0E	00																						
	0F	00																						
	10	00																						
	11	00																						
	12	00																						
	13	00																						
	14	00																						
	15	00																						
	16	00																						
	17	00																						
	18	00																						
	19	00																						
	1A	00																						
	1B	00																						
	1C	00																						
	1D	00																						
	1E	00																						
	1F	00																						
	20	00																						
	21	00																						
	22	00																						
	23	00																						
	24	00																						
	25	00																						
	26	00																						
	27	00																						
	28	00																						
	29	00																						
	2A	00																						
	2B	00																						
	2C	00																						
	2D	00																						
	2E	00																						
	2F	00																						

Capítulo 6

Implementación de un puente UART/*ZigBee*

6.1. Hardware empleado

Uno de los productos finales del presente documento, tal y como se ha descrito en el capítulo 1 es el de, dado un microcontrolador CC2530, generar un código que permita una implementación (en este caso será en estrella) que haga posible la comunicación entre un SoC *Coordinador* (por ejemplo, conectado a un PC) y una serie de *End Devices* que estén conectados vía UART, por ejemplo, a los microcontroladores de robots o de otros dispositivos. En realidad el CC2530 hace de puente ciego, de forma que el dispositivo conectado vía UART al SoC genera una trama de 32 octetos tal y como se ha descrito en la sección 5.2 y se pasa via UART al CC2530. Éste se encarga de extraer la dirección de destino (2 bytes) y el *End Point* de destino, y de rellenar la información propia del CC2530 que deba haber en la trama. Luego esta trama se envía por *ZigBee*. Al revés, cuando se recibe la trama vía *ZigBee* ésta se pasa via UART al dispositivo final, de forma que éste extrae de la trama los datos que necesita para realizar los cálculos que se precisen.

En el desarrollo se ha empleado el CC2530 en su forma CC2530EM, módulo que se inserta en una placa SmartRF05EB. El hardware final a implementar sobre un dispositivo final arbitrario girará alrededor del diseño del CC2530EM, tal y como se describe en la sección 8.2 se puede adaptar el SoC a una placa que se inserte en los puertos serie de los dispositivos finales. En cualquier caso, para el desarrollo del software se ha empleado para la conexión al PC un cable adaptador Serie RS-232 a USB, y este USB reconocido como USB serie por una consola bajo GNU/Linux se ha empleado para monitorizar la llegada de tramas o para enviarlas por UART al CC2530 para que éste procediera al envío por *ZigBee*.

6.2. Descripción del software empleado

La inmensa mayoría del trabajo presentado en este documento se ha invertido en la programación de un puente UART/ZigBee de forma que, respetando la especificación de la trama de 32 octetos, el sistema usuario usa *ZigBee* como si fuera una conexión serie estándar.

La programación del software se ha realizado sobre el Z-Stack de Texas Instruments. La estructura de un proyecto tipo incluye todas las referencias a las librerías a utilizar (ver capítulo 4) con los ajustes necesarios para habilitar el software a emplear. En particular cabe destacar que en *Options/Preprocessor* hay que sustituir

Listing 6.1: Opciones preprocesador para debugging

```
1 LCD_SUPPORTED=DEBUG
```

por

Listing 6.2: opciones preprocesador en producción

```
1 LCD_SUPPORTED
```

Porque de lo contrario la placa vuelca por el puerto RS-232 la salida que en principio va al LCD.

En el software que emplea la placa SmartRF05EB no es necesario deshabilitar ningún periférico, pero es necesario habilitar el puerto serie cambiando el siguiente código en **hal_board_cfg.h**:

Listing 6.3: Configuración para habilitar el puerto serie

```
1
2 /* Set to TRUE enable UART usage, FALSE disable it */
3 /*#ifndef HAL_UART
4 #if (defined ZAPP_P1) || (defined ZAPP_P2) || (defined ZTOOL_P1) || (defined
    ZTOOL_P2)
5 #define HAL_UART TRUE
6 #else
7 #define HAL_UART FALSE
8 #endif
9 #endif*/
10
11 #define HAL_UART TRUE
```

para evitar problemas. Se usará el puerto 0 sobre los pines 16 y 17, tal y como se muestra en la figura 8.2.

Básicamente, como se ha programado empleando el OSAL, se programa como respuesta a eventos. En realidad hay tres eventos, de entre los que notifica el sistema operativo, que son de interés. Estos eventos se gestionan en el *event handler* correspondiente.

Llegada de un mensaje por *ZigBee* : el mensaje se gestiona en la rutina

Listing 6.4: Prototipo de la rutina de gestión de un mensaje OTA/ZigBee entrante

```
1 UINT16 GenericApp_ProcessEvent( byte task_id, UINT16 events )
```

y dentro de ella, en la parte

Listing 6.5: Código para la gestión del mensaje OTA/ZigBee entrante

```
1 switch ( MSGpkt->hdr.event )
2 {
```

```
3
4           ...
5           case AF_INCOMING_MSG_CMD:
6
7   #if HAL_LCD == TRUE
8           mostrarcomando(MSGpkt);
9   #endif
10  //  envia mensaje por uart
11            enviarmensajeUART(MSGpkt);
12            break;
13          ...
14  }
```

esta implementación muestra como ante el evento que el OSAL describe como AF_INCOMING_MSG_CMD este mesaje se procesa empleando la rutina *enviarmensajeUART(MSGpkt)*, la cual extrae el mensaje entrante de la trama *ZigBee* y lo envía por UART bajo la configuración del puerto 0 definida anteriormente. El código en cuestión es:

Listing 6.6: Canalización del mensaje entrante OTA/ZigBee al puerto serie

```
1  void enviarmensajeUART(afIncomingMSGPacket_t *pkt)
2  {
3          if(pkt->cmd.DataLength == LONGITUD_TRAMA)
4          {
5                  HalLedSet (HAL_LED_3, HAL_LED_MODE_TOGGLE);
6                  HalUARTWrite(SERIAL_APP_PORT, pkt->cmd.Data, LONGITUD_TRAMA);
7          }
8  }
```

que se limita a escribir en el puerto UART SERIAL_APP_PORT un número de bytes igual a LONGITUD_TRAMA.

Llegada de un mensaje por UART : la definición del *event handler* para la recepción por UART es algo distinta, pues se define en las rutinas de inicialización del OSAL:

Listing 6.7: Configuración del puerto UART y del gestor de eventos (SerialApp$_{CallBack}$)

```
1  void GenericApp_Init( byte task_id )
2  {
3          ...
4          /* B - SERIAL APP PART */
5
6           halUARTCfg_t uartConfig;
7
8          SerialApp_TaskID = task_id;
9          SerialApp_RxSeq = 0xC3;
10
11         uartConfig.configured           = TRUE;
12         uartConfig.baudRate             = SERIAL_APP_BAUD;
13         uartConfig.flowControl          = TRUE;
14         uartConfig.flowControlThreshold = SERIAL_APP_THRESH;
15         uartConfig.rx.maxBufSize        = SERIAL_APP_RX_SZ;
16         uartConfig.tx.maxBufSize        = SERIAL_APP_TX_SZ;
17         uartConfig.idleTimeout          = SERIAL_APP_IDLE;
18         uartConfig.intEnable            = TRUE;
19         uartConfig.callBackFunc         = SerialApp_CallBack;
```

```
20
21          HalUARTOpen (SERIAL_APP_PORT, &uartConfig);
22
23          ...
24  }
```

en la línea

Listing 6.8: Linea de la asignación del gestor de eventos para UART entrante

```
1  uartConfig.callBackFunc           = SerialApp_CallBack;
```

se indica el *event handler* de la llegada de un mensaje por UART. *SerialApp_CallBack* está implementada de la siguiente forma:

Listing 6.9: Implementación de SerialApp_CallBack

```
1  static void SerialApp_CallBack(uint8 port, uint8 event)
2  {
3
4
5          (void)port;
6          HalUARTRead(SERIAL_APP_PORT, SerialApp_TxBuf, LONGITUD_TRAMA);
7          HalLedSet (HAL_LED_4, HAL_LED_MODE_TOGGLE);
8
9          // verifica la integridad de la trama chequeando el primer y el
10         // ultimo octeto
11         if((SerialApp_TxBuf[0] == 0xFF) && (SerialApp_TxBuf[31] == '\0'))
12         {
13                 //  rellena el buffer correcto
14                 for(int i = 0;i < LONGITUD_TRAMA;i++)
15                 {
16                         mensajeAenviar32bytes[i] = SerialApp_TxBuf[i];
17                 }
18                 // envia la variable global mensajeAenviar32bytes
19                         GenericApp_MessageMSGCBSCA();
20                         HalLedSet (HAL_LED_3, HAL_LED_MODE_TOGGLE);
21                 }
22         }
```

tras la verificación y tras completar el buffer definido como variable global mensajeAenviar32bytes[] se invoca la función *GenericApp_MessageMSGCBSCA()*.

Listing 6.10: Función de envío GenericApp_MessageMSGCBSCA()

```
1  void GenericApp_MessageMSGCBSCA(void)
2  {
3          GenericApp_DstAddr.addr.shortAddr
4          = (uint16)(mensajeAenviar32bytes[4] & 0x00ff) |
5          ((mensajeAenviar32bytes[5] << 8) & 0xff00 );
6          //  A donde env a ...
7          HalLcdWriteStringValue("Destino: ",
8          GenericApp_DstAddr.addr.shortAddr,16,3);
9
10         //  LSB
11         mensajeAenviar32bytes[1] = (uint8)(NLME_GetShortAddr());
```

```
12          //  MSB
13          mensajeAenviar32bytes[2] = (uint8)(NLME_GetShortAddr() >> 8);
14          //  EP de origen
15          mensajeAenviar32bytes[3] = GENERICAPP_ENDPOINT;
16
17          //  Orden de env o ...
18          AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc,
19                                      GENERICAPP_ENDPOINT,
20                                      LONGITUD_TRAMA,
21                                      (byte *)&mensajeAenviar32bytes,
22                                      &GenericApp_TransID,
23                                      AF_DISCV_ROUTE, AF_DEFAULT_RADIUS );
24  }
```

de esta forma se rellenan los bytes 1,2 y 3 (propios del dispositivo desde dónde se envía, y que no se rellenan en el microcontrolador o PC desde dónde se ha enviado la trama), y esta se envía a **GenericApp_DstAddr.add.shortAddr** y siempre al endpoint GENERICAPP_ENDPOINT que ha sido definido como 10 o 0x0A en este módulo *ZigBee* y este es el valor que se debe usar. En la trama se mantiene el espacio reservado por si se decidiera en un futuro modificar el código y permitir enviar a diferentes endpoints y a diferentes clusters, pero en esta aplicación no es necesario.

Evento disparado por temporizador : finalmente, y sobre todo para la fase de verificación[1]. En cualquier caso, se ha propuesto un evento disparado por tiempo que se activa cada 30 segundos. La activación se realiza en

Listing 6.11: Gestor de eventos disparados por tiempo

```
1  void GenericApp_Init( byte task_id )
2  {
3          ...
4          osal_start_timerEx( GenericApp_TaskID,
5                  GENERICAPP_SEND_MSG_EVT,
6                  GENERICAPP_SEND_MSG_TIMEOUT );
7                  ...
8  }
```

este evento se gestiona en

Listing 6.12: Código gestion evento disparado por temporizador

```
1  UINT16 GenericApp_ProcessEvent( byte task_id, UINT16 events )
2  {
3          ...
4
5          // Send a message out - This event is generated by a timer
6          //  (setup in GenericApp_Init()).
7          if ( events & GENERICAPP_SEND_MSG_EVT )
8          {
9
10 #ifdef  PRUEBA_SUPPORT
11                 GenericApp_SendTheMessageNEW();
```

[1]en el código final no se implementa porque lo único que se quiere es un puente transparente, respetando la necesidad de rellenar información referente a la dirección, endpoint y cluster de destino

```
12 #endif
13
14                     return (events ^ GENERICAPP_SEND_MSG_EVT);
15           }
16
17           ...
18 }
```

6.3. Terminal Serie en PC

La conexión física entre las placas *SmartRF05EB* con el *CC2530EM* se realiza usando un cable conversor RS232/USB, de forma que en el PC se muestra como un puerto serie. En particular en GNU/Linux aparece como /dev/ttyUSB0[2].

Para la comunicación con un PC se han escrito dos programas en C, uno que envía y otro que recibe[3] tramas como la propuesta en la sección 5.2. Estos programa se ha realizado bajo GNU/Linux, puesto que el manejo de puertos bajo este sistema operativo es mucho más sencillo que bajo otros, como Windows.

Existen dos formas[5] de estructurar la anterior programación

1. Si el mismo programa va a tomar decisiones basadas en la recepción de mensajes por el puerto serie, es necesario programar un único proceso con una serie de hilos o *threads*. Aparte de main(), haría falta:

   - Un primer *thread* que **lee del puerto serie**, y escribe en una variable global al proceso. La escritura en esa varible global (*bufferrecibido[][]*) debe ser definida como *sección crítica* para evitar la *condición de carrera*. El acceso al puerto serie no genera este problema puesto que la lectura y escritura se realizan sobre canales distintos.
   - Un segundo *thread* que **escribe en el puerto serie** lo que se va escribiendo en otra variable global (*bufferaenviar[][]*). La lectura de esta variable puede producir la *condición de carrera*, por lo que la lectura también debe ser definida como sección crítica.
   - Uno o más *threads* que leen de (*bufferrecibido[][]*) y escriben en (*bufferaenviar[][]*) también definiendo estos accesos como *sección crítica*, para evitar la *condición de carrera*. Estos *threads* realizarían los cálculos y tomarían las decisiones que inyectarían de forma asíncrona en la red.

   El anterior esquema define la estructura de un proceso que gestiona una red de sensores en un sistema distribuído. La API C de POSIX permite hacerlo, como también las APIs de otros lenguajes, como Java. Una buena referencia sobre POSIX es [23].

[2] Si hubiera más conexiones USB/Serie aparecería como /dev/ttyUSBx donde x sería un número

[3] los mensajes entrantes al PC se pueden leer con una terminal, por ejemplo *putty*, abriendo el puerto serie, que normalmente en GNU/Linux[4] es /dev/ttyUSB0

[5] En cualquier caso y en GNU/Linux para abrir el puerto USB/Serie al código lo más sencillo es ejecutar *putty* previamente

2. Si lo que se pretente es monitorizar la llegada de mensajes y definir *manualmente* el envío de mensajes, a efectos de verificación de funcionamiento del sistema, y puesto que la lectura/escritura de dos procesos sobre el mismo puerto serie no produce problemas de *condición de carrera*, la estructura más sencilla que cumpliera con estas especificaciones sería la siguiente:

   - Un primer **proceso** para ir leyendo del puerto serie a medida que llegan los mensajes
   - Un segundo **proceso** para escribir en el puerto serie cuando el usuario decidiera hacerlo.

De esta forma habría **dos procesos**. En el puerto serie, al haber diferentes canales para la lectura y la escritura, no se produciría la *condición de carrera*. En el caso del proyecto objeto del presente documento se ha implementado el código siguiendo ambos paradigmas: El **primero** se emplea cuando se quiere un sistema completo que lea y envíe desde el mismo proceso y tome decisiones. El **segundo** se emplea cuando se quiere realizar una verificación rápida del funcionamiento de una serie de dispositivos. Ambos se han implementado a través del uso de librerías en C programadas con las funciones de lectura y envío. El código de estas librerías se encuentra en **functionswriteinserial.c** y eb **escribiryenviar.c** y el .h en **writeinserial.h**. A partir de este código se ha compilado la librería **zigbeeti** en el .so **libzigbeeti.so.1.0** y sus *symbolic links* **libzigbeeti.so** y **libzigbeete.so.1**, y se han instalado en /usr/lib. Los prototipos de las funciones contenidas en esta librería son las siguientes:

Listing 6.13: Prototipos funciones contenidas en la librería libzigbeeti

```
1
2  //      FUNCION
3  //      cada vez que se invoca devuelve UNA trama de LONGITUDBUFFERSERIE octetos
4  //      as  que cada vez que para recibir constantemente hay que meter esta
       funci n
5  //      en alguna estructura iterativa
6  //      puertoserie             cadena del tipo /dev/ttyUSB0"
7  //      pmensajearecibir        puntero de tipo unsigned char*
8  //                              con LONGITUDBUFFERSERIE bytes reservados
9  //                              en la llamada.
10 void zigbeeti_recibir(char* puertoserie, unsigned char* pmensajearecibir)
11
12 //      FUNCION
13 //      se usa para generar una trama de 32 octetos a enviar, suministrando la
       siguiente informaci n
14 //      unsigned int direcciondestino   --      direcci n de destino que se
       convertir  en 2 bytes
15 //      unsigned int endpointdestino    --      endpoint (1 byte) de destino
16 //      unsigned int clusterdestino     --      cluster de destino, en caso de
       usarse (1 byte)
17 //      unsigned int comando            --      comando (2 bytes)
18 //      char* cadenaasciidatos          --      puntero a un string que encapsula
       los datos a enviar
19 //      char* puertoserie               --      puntero a un string que describa
       el puerto a abrir
20 void zigbeeti_enviar(unsigned int direcciondestino, unsigned int endpointdestino,
21         unsigned int clusterdestino, unsigned int comando,
22         char* cadenaasciidatos,char* puertoserie)
23
24 //      FUNCION
```

```
25 //      opens the PUERTOSERIEOBJETIVO defined as a chain in writeinserial.h
26 //      blocking function
27 int zigbeeti_open_port_bloqueante(char* puertoserie)
28
29 //      FUNCION
30 //      Rellena el buffer variable global a enviar serie CON LOS DATOS
       SUMINISTRADOS
31 //      pmensajeaenviar         puntero unsigned char* con LONGITUDBUFFERSERIE
       bytes libres
32 void zigbeeti_rellenaBufferConInfo(unsigned int entrada_direcciondestino ,
33         unsigned int entrada_endpointdestino , unsigned int entrada_clusterdestino
               ,
34         unsigned int entrada_comando , char* entrada_datos,
35         unsigned char* pmensajeaenviar)
36
37 //      FUNCION
38 //      Imprime los datos interpretados ANTES de abrir el puerto
39 void zigbeeti_verificaloqueseenvia(char* mensajeaenviar)
```

6.3.1. Dos procesos independientes para lectura/escritura

La implementación de un proceso para lectura y otro para escritura se realiza empleando los siguientes .c y .h:

- writeinserial.c
- readinserial.c
- writeinserial.h [6]

preparados para generar dos ejecutables:

- writeinserial[7]
- readinserial[8]

El **proceso de escritura** se ha implementado con el ejecutable **writeinserial.c** que tiene la siguiente estructura:

Listing 6.14: Código en writeinserial.c

```
1 int main(int argc, char *argv[])
2 {
3
4         char entrada_dispositivo[MAXIMALINEA]; //      esperamos algo como "
              ttyUSB0"
5         unsigned int entrada_direcciondestino; //      esperamos dos bytes
6         unsigned int entrada_endpointdestino;  //      esperamos un byte
7         unsigned int entrada_clusterdestino;   //      esperamos un byte
```

[6] Para emplear las librerías compiladas anteriormente que se suponen instaladas correctamente
[7] gcc -o writeinserial writeinserial.c -lzigbeeti
[8] gcc -o readinserial readinserial.c -lzigbeeti

```
8           unsigned int entrada_comando;           //      esperamos dos bytes
9           char entrada_datos[MAXIMALINEA];        //      esperamos entre 0 y 20
                bytes
10
11          printf("Dispositivo a conectar ,por ejemplo /dev/ttyUSB0):");
12          scanf("%s" , entrada_dispositivo);
13
14          printf("Direcci n de destino en HEXADECIMAL (2 bytes):\n");
15          scanf("%x" , &entrada_direcciondestino);
16
17          printf("End Point de destino en HEXADECIMAL (estamos usando 10, en
                HEXADECIMAL 0x0A):\n");
18          scanf("%x" , &entrada_endpointdestino);
19
20          printf("Cluster de destino en HEXADECIMAL (por ahora no se usa):\n");
21          scanf("%x" , &entrada_clusterdestino);
22
23          printf("Comando a enviar en HEXADECIMAL (2 bytes):\n");
24          scanf("%x" , &entrada_comando);
25
26          printf("Datos a enviar, por ahora caracteres (20 m ximo) y sin espacios:\
                n");
27          scanf("%s" , entrada_datos);
28
29          //      FUNCION LIBRERIA ZIGBEETI
30          //
31
32          zigbeeti_enviar(entrada_direcciondestino, entrada_endpointdestino,
33                  entrada_clusterdestino, entrada_comando, entrada_datos ,
34                  entrada_dispositivo);
35
36          return 0;
37          }
38
39
40  }
```

Y el **proceso de lectura** se ha implementado con el ejecutable **readinserial.c** que tiene la siguiente estructura:

Listing 6.15: Código en readinserial.c

```
1   main()
2   {
3
4           //      Local variables
5           //      unsigned char pmensajearecibir[LONGITUDBUFFERSERIE];
6           unsigned char pmensajearecibir[LONGITUDBUFFERSERIE];
7           char entrada_dispositivo[MAXIMALONGITUDNOMBREPUERTOSERIE];
8           int i;
9
10          printf("Dispositivo a conectar ,por ejemplo /dev/ttyUSB0):");
11          scanf("%s" , entrada_dispositivo);
12
13
14          zigbeeti_recibir(entrada_dispositivo , pmensajearecibir);
15
16          //      escribir lo que se ha recibido
17          for(i=0;i < LONGITUDBUFFERSERIE; i++)
```

```
18          {
19                  printf("%2.2x",*(pmensajearecibir + i));
20          }
21          printf("\n");
22  }
```

6.3.2. Un único proceso de lectura/escritura/toma de decisiones implementado con hilos y secciones críticas

La estructura propuesta en la la sección 6.3.1 es insuficiente si se pretende disponer de un código que de forma integrada realice varias tareas simultáneas que incluyan la lectura/recepción del puerto USB/Serie, la escritura/envío al puerto serie y una parte del código que realice la toma de decisiones. La arquitectura del código requiere la comunicación entre distintas partes de un código que actúan de forma concurrente sobre una serie de variables compartidas. Por ello se ha optado por programar una aplicación en C utilizando hilos de POSIX. Para ello se ha empleado la librería **pthread** cuyo .h es **pthread.h**, y que define las funciones necesarias para implementar hilos en C bajo POSIX, soportado por GNU/Linux.

La figura 6.1 describe la forma de acceso asíncrona a las variables compartidas, y las diferentes relaciones entre los hilos

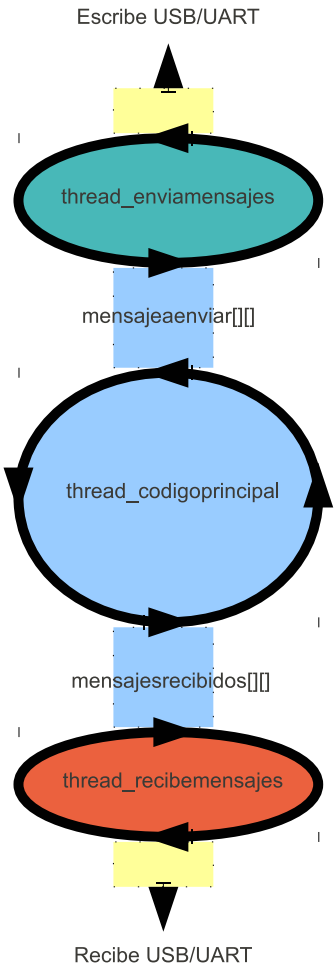

Figura 6.1: Acceso asíncrono de hilos a variables compartidas en **escribiryenviarconhilos.c**

La figura 6.1 muestra la existencia de una serie de variables globales compartidas que sirven para que los hilos se comuniquen. En realidad hay dos grupos:

Variables de datos: son aquellas que recogen datos globales que se reciben o envían

- unsigned char mensajesrecibidos[][]
- unsigned char mensajesaenviar[][]

- char entrada_dispositivo[]
- unsigned int entrada_direcciondestino
- unsigned int entrada_endpointdestino

Variables de control: son aquellas que llevan la cuenta de algún proceso

- int nmensajesrecibidos
- int nmensajesaenviar
- int nmensajesefectivamenteenviados
- int nmensajesefectivamenteprocesados

y los hilos o *threads* empleados son

thread_recibemensajes recibe mensajes entrantes del puerto serie definido por **entrada_dispositivo[]** y los va colocando en **mensajesrecibidos[][]**

thread_enviamensajes toma los mensajes a enviar de **mensajesaenviar[][]** y los envía por el puerto serie definido por **entrada_dispositivo[]**

thread_codigoprincipal lee de **mensajesrecibidos[][]** los mensajes entrantes y coloca en **mensajesaenviar[][]** las respuestas. Este hilo implementa las tareas de control necesarias para cumplir con el objetivo de la programación. En el caso particular del código implementado hace lo siguiente:

- Espera a que el dispositivo/robot envíe un mensaje de identificación, con comando 0x0000, para poder capturar la dirección de destino de los mensajes siguientes
- Empieza a enviar órdenes y argumentos, y para ello espera que el dispositivo/robot de destino interrumpa sus mensajes de identificación periódicos y comience a enviar mensajes de estado al finalizar la tarea propuesta.
- Cuando recibe un mensaje de confirmación envía la orden siguiente.

El anterior esquema es relativamente sencillo, pero la complicación radica en lo que se espera que haga **thread_codigoprincipal**. El código del mismo[9] se encuentra en el anexo B, y éste realiza, en este caso particular, las siguientes tareas:

- Espera a que en mensajesrecibidos[][] llegue algún mensaje[10], y cuando llega extrae la información del mismo, en particular la dirección y *end point* de origen, para que se le pueda responder. En realidad es una versión simplificada de los catálogos de servicios que se generarían si se empleara un paquete de software para gestionar por *multiagentes* la comunicación en una red más compleja con diferentes servicios en diferentes nodos. En la referencia [18] se puede encontrar información sobre *JADE*.
- Una vez adquirida la dirección de destino al haberse identificado el dispositivo/robot, el PC está listo para iniciar la secuencia de órdenes. Cada orden se procesa de la siguiente manera:

[9] Parcialmente recortado para omitir las secciones que se repiten con una estructura similar

[10] Se asume que el dispositivo con el que se debe conectar envía al *Coordinador* un mensaje periódico indicando, al menos, su dirección en la red y su *end point*

1. Se encola un mensaje con el comando y los parámetros correctamente incluidos en la trama de 32 octetos.
2. Se espera la respuesta del dispositivo remoto.
3. Cuando la respuesta llega (procesando los mensajes colocados en el buffer mensajesrecibidos[][]) se identifica el comando de respuesta y los parámetros del mismo. En este caso particular no se analizan estos datos, pero es perfectamente posible definir un comportamiento más complejo basado en el estado del dispositivo remoto expresado en el mensaje entrante, que informa del resultado de la petición de ejecución de la orden transmitida.
4. Se pasa a la orden siguiente, si existe, y se repite lo anterior

Hay un elemento más a considerar. Tal y como se ha comentado anteriormente, y se ha representado gráficamente en la figura 6.1, existen una serie de variables globales compartidas que están expuestas a la *condición de carrera*. Para evitarlo se ha empleado un mutex que protege las secciones críticas del código. De esta forma se garantiza que el acceso a las variables globales compartidas se ejecuta de forma atómica por las diferentes porciones de código que acceden a las mismas en los diferentes hilos.

6.4. Funcionamiento del sistema

El comportamiento esperado del par control(PC)-dispositivo remoto(Robot) implica que se introducirá la configuración necesaria para que el sistema de dos (o más) dispositivos se comporte respetando las siguientes especificaciones[11]:

- **En el dispositivo remoto/robot:**

  1. Inicia el envío del mensaje periódico (con comando 0x0000) hasta que haya una respuesta por parte del control/PC
  2. Ante el primer mensaje entrante, se deja de enviar el mensaje periódico y se ejecuta la orden de acuerdo al comando y parámetros del mensaje entrante
  3. Al finalizar la ejecución de la orden, se envia al control/PC un mensaje de estado, que contendrá el comando 0x000B genérico que se interpreta como '*La tarea se ha completado con éxito*' si la tarea se ha completado con éxito
  4. El dispositivo remoto/robot queda a la espera de la entrada de un nuevo mensaje con una nueva orden.

- **En el control/PC:**

  1. Se espera que lleguen mensajes de dispositivos remotos (en este caso sólo uno) para conocer su dirección y end point.
  2. Una vez llega el primer mensaje con el comando 0x0000 y se extraen la dirección de origen y el end point de origen, se envía el primer mensaje con el comando y parámetros adecuados

[11] La selección del software y hardware se describe en la sección 6.5

3. Cuando llega la respuesta de finalización exitosa (normalmente comando 0x000B), se envia la siguiente orden, y se repite este proceso hasta que se agotan las órdenes a comunicar

Lo anterior se podría resumir en el siguiente pseudocódigo:

Algoritmo 6.4.1: DISPOSITIVOREMOTO($void$)

Inicia envío mensaje periódico
while Sin respuesta
  do Persiste en el envío
if Llega un mensaje
  then {
    repeat
      Obten comando y parámetros
      Ejecuta la orden
      Envío mensaje de finalización al control
      Espera la llegada de un nuevo mensaje
    until indefinidamente
  }

Algoritmo 6.4.2: CONTROL($void$)

Espera la llegada de un mensaje de identificación
if Llega un mensaje de identificación
  then Obten la dirección y end point de destino
repeat
  {Enviar mensaje con comando y parámetros correctos
  {Esperar la llegada del mensaje de confirmación de ejecución
until indefinidamente

6.5. Verificación del funcionamiento del sistema

Para la verificación del funcionamiento del sistema se ha empleado el siguiente montaje:

PC1-SmartRB05EB-CC2530/Coordinador: en él el CC2530 está programado con el código C puente descrito en el anexo A, configurado como *coordinador*. El PC está equipado con el código descrito en el anexo B.2. Su funcionamiento exacto ya se ha descrito en la sección 6.3.2. El PC1 se conecta a la placa por un cable conversor serie/USB.

PC2-SmartRB05EB-CC2530/End Device: en él el CC2530 está programado con el código C puente descrito en el anexo A, configurado como *end devide*. El PC1 se conecta a la placa por un cable conversor serie/USB. El PC ejecuta dos procesos que acceden al USB/Serie, uno para lectura y otro para escritura, de forma que por un terminal se envían mensajes a 0x0000 y por otro se reciben. Esto simula el comportamiento de un robot que recibe órdenes y tarda algún tiempo en confirmar la ejecución. Cuando ésta ha concluido el PC1 envía la siguiente orden, hasta que la secuencia de órdenes concluye.

El montaje de verificación se puede ver en la figura 6.2.

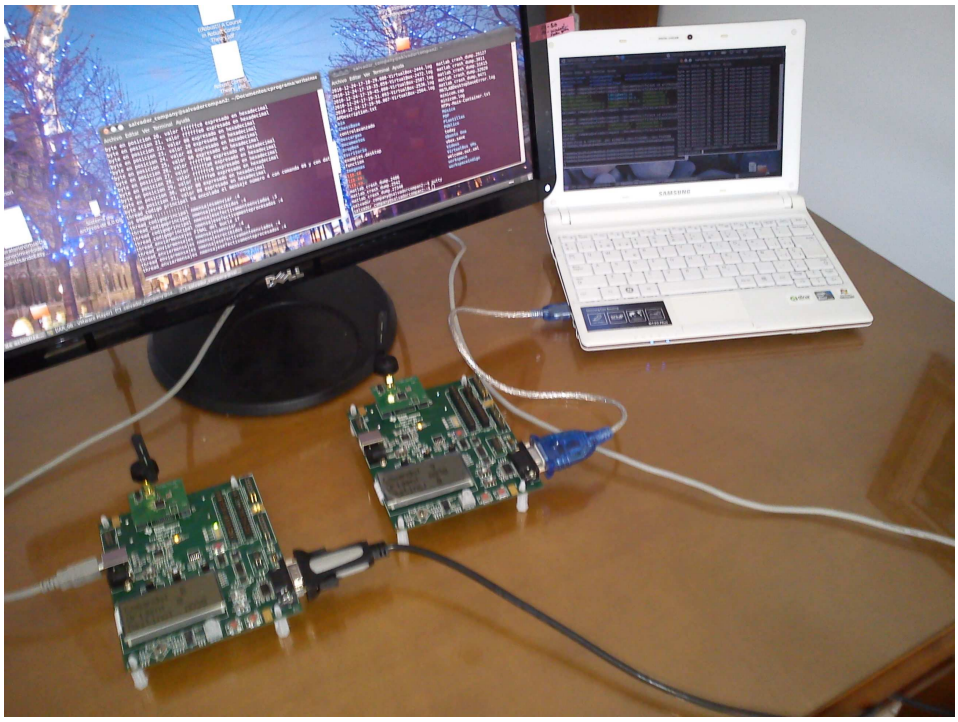

Figura 6.2: Montaje para la verificación del correcto funcionamiento del software tanto del CC2530 como del PC, tanto como coordinador/PC1 como end device/dispositivo remoto/PC2

Tras enviar desde el PC2 el mensaje de identificación con comando 0x0000, el PC1 respondía correctamente enviando los 6 mensajes de prueba al origen del mensaje de indentificación inicial, y esperando para cada mensaje enviado a la confirmación del dispositivo remoto.

Capítulo 7

Aplicación al control de *Robotino*

7.1. Descripción de *Robotino*

Robotino es un sistema móvil de aprendizaje e investigación en el campo de la mecatrónica desarrolado por *FESTO DIDACTIC*, Dekendorf, Alemania, en cooperación con la *Robotics Equipment Corporation* (REC), en Munich, Alemania. Este sistema se emplea en todo el mundo.

El chasis de *Robotino* es metálico y protegido contra colisiones por una banda de goma que a su vez dispone de un sensor. Las dimensiones del mismo son 370 mm de diámetro y 210 mm de altura, con un peso total de unos 11 kg. Con sus ruedas omnidirecionales, *Robotino* puede moverse hacia adelante, hacia atrás y también lateralmente. Tres motores industriales de corriente continua con encoders y piñones intercambiables permiten velocidades de hasta 10 km/h. El chasis contiene nueve sensores infrarrojos de distancia. También se pueden instalar sensores inductivos u ópticos reflectivos. Adicionalmente se puede instalar una webcam a color con compresión JPEG, cuyas imágenes pueden enviarse a un PC remoto via LAN inalámbrica para evaluación o simple visualización. La alimentación del sistema se consigue gracias a dos baterias de plomo-gel que permiten una operación autónoma de hasta dos horas.

Robotino dispone de una amplia gama de accesorios como sensores (opticos, inductivos, cámaras, giróscopos, medidores de distancia láser, ...) así como accesorios de manipulación (brazos articulados, elevadores,...), etc. Además, existen interfaces que permiten integrar los dispositivos propios de los usuarios en el sistema *Robotino*.

La figura 7.1 muestra el aspecto básico del *Robotino*.

Figura 7.1: *Robotino*

En cuanto a su arquitectura, *Robotino* dispone de un PC empotrado con un procesador *AMD Geode* con un kernel de *real-time Linux*. El sistema operativo, programas y datos se almacenan en una tarjeta flash. El PC empotrado dispone de varias interfaces, como Ethernet, LAN inalámbrica, USB, RS232 y un puerto VGA.

El servidor de *Robotino*, una aplicación para *real-time Linux*, es la pieza clave del sistema. Controla los motores del *Robotino* y se puede comunicar con aplicaciones externas de tres formas diferentes:

1. Una librería de C++ para Linux con funcionas básicas se encuentra disponible para la programación del PC empotrado del *Robotino*.
2. Una interfaz TCP/IP para la comunicación con el PC externo vía LAN inalámbrica. De esta forma, el usuario tiene la opción de escribir aplicaciones en C++ para el control inalámbrico de *Robotino* empleando las funciones de la librería C++ de Windows
3. *Robotino View*, un entorno de programación gráfico que se comunica desde el PC de control con el *Robotino* a través de la LAN inalámbrica, sin necesidad de compilar o descargar código en el PC empotrado en el *Robotino*.

La activación de los tres motores se consigue gracias a un panel adicional con entradas y salidas analógicas y digitales que se conecta al PC empotrado y embarcado en el *Robotino* a través de una conexión serie. El panel de control dispone de los siguientes puertos para potenciales expansiones:

- Ocho entradas analógicas de 0-10 V.
- Ocho entradas/salidas digitales a 24 V.
- Dos relés para actuadores adicionales

En cuanto a la programación, *Robotino View* desarrollado por REC, soporta el desarrollo de secuencias de control basadas en el lenguaje estandarizado de bloques de acuerdo a la especificación IEC 61131. Los usuarios disponen de un módulo de navegación que incluye funciones para definir movimiento y caminos, así como librerías matemáticas y lógicas. Asimismo más librerías se pueden desarrollar programando en C++ o en Lua. Respecto a la comunicación de varios *Robotinos* entre sí y con controladores externos, existen bloques que permiten la comunicación vía TCP/IP y UDP. Además, existen numerosas APIs que permiten al usuario programar *Robotino* en diferentes lenguajes de alto nivel, como C++, .Net, JAVA, LabVIEW, Matlab y otros.

Para más detalles se puede consultar las referencias [25] y [26].

7.2. Conexión física entre el *SmartRF05EB* y *Robotino*

Tal y como se ha explicado en la sección 7.1, *Robotino* dispone de una conexión USB que puede ser empleada como conexión del cable USB/Serie que se ha empleado hasta ahora para la comunicación entre el conjunto *SmartRF05EB/CC2530* con cualquier dispositivo que tuviera un puerto USB y reconociera el mismo como USB/Serie. En particular, se ha empleado un PC con Linux con algún tipo de código para la lectura/escritura, tal y como se ha descrito en la sección 6.2. Para interactuar con el *Robotino* la configuración debe ser la siguiente:

PC de control: será un PC con el código descrito en la sección 6.3.2, conectado vía cable USB/Serie a un *SmartRF05EB/CC2530* programado con el código descrito en la sección 6.2 compilado como **coordinador**.

Robotino: el *Robotino* se conectará vía cable USB/Serie a un *SmartRF05EB/CC2530* programado con el código descrito en la sección 6.2 compilado como **end device**. El software implementado en el *Robotino* se ajustará al algoritmo *control* descrito en la sección 7.1.

7.3. Secuencia de operaciones

Para la demostración final se ha enviado al *Robotino* la siguiente secuencia de operaciones[1]:

[1] En resultado de la ejecución se puede ver en el video que se adjunta al presente documento

Traslación absoluta a 015000:000000:000000 : *Robotino* se encuentra en $(x, y, z) = (0, 0, 0)$ en metros, y se envía a $(x, y, z) = (1.5, 0, 0)$, en metros, en un sistema de coordenadas absoluto.

Rotación relativa de 090000:::::::::::::: : *Robotino* gira 90º en sentido antihorario en la posición en la que se encuentra.

Traslación absoluta a 015000:001000:000000 : *Robotino* avanza desde la posición actual $(1.5, 0, 0)$ a $(x, y, z) = (1.5, 1.0, 0)$. Puesto que el anterior paso lo ha orientado adecuadamente, simplemente ha de avanzar un metro.

Rotación relativa de 090000:::::::::::::: : *Robotino* gira 90º en sentido antihorario en la posición en la que se encuentra. De esta forma ya acumula desde su posicion inicial una rotación absoluta de 180º.

Posicionamiento relativo al chasis de la pinza en 000200:000100:000000 : *Robotino* posiciona su pinza a 20 cm de su posición de referencia y 10 cm por encima de su plano de referencia.

Traslación absoluta a 000000:001000:000000 : finalmente *Robotino* avanza desde $(1.5, 1.0, 0)$ a $(x, y, z) = (0, 1.0, 0)$.

La anterior secuencia simula una aproximación a un objeto, su captura mediante un brazo robótico embarcado y su entrega al final del recorrido. Estas órdenes se encuentran en el hilo **thread_codigoprincipal** del fichero **funcioneswriteinserialhilos.c**.

7.4. Resultado de la ejecución

El resultado de la ejecución de la secuencia de órdenes descritas en la sección 7.3 se puede ver en el video adjunto. Aunque *Robotino* exhibe ciertas particularidades en cuanto al reconocimiento de puertos USB/Serie, la ejecución ha sido satisfactoria y ha permitido una demostración práctica de la utilidad de la implementación del módulo *ZigBee*, en este caso

Capítulo 8

Implementación final propuesta

8.1. Justificación para una implementación definitiva distinta

Tal y como se ha descrito en la sección 3.1, en el caso de integrar el CC2530 en una placa junto con un *SoC* de control no parece práctico la conexión de ambos vía RS232/USB, tal y como se ha procedido durante el desarrollo del puente *ZigBee*/UART. En cambio, se opta por una configuración similar a la mostrada en la figura 8.1, en la que la entrada/salida UART se consigue empleando los pines 16 y 17 para lectura/escritura, así como la conexión de los mismos al puerto serie del microcontrolador del sistema via conexión directa.

8.2. Adaptación del hardware

En las figuras 8.1 y 8.2 se puede ver el diseño esquemático propuesto para el *weelrobot* y basado en el propuesto por Texas Instruments que desarrolla el esquema del CC2530EM, que integra en una única placa el CC2530 y todos los componentes necesarios para que se puedan emplear las funciones de radiofrecuencia.

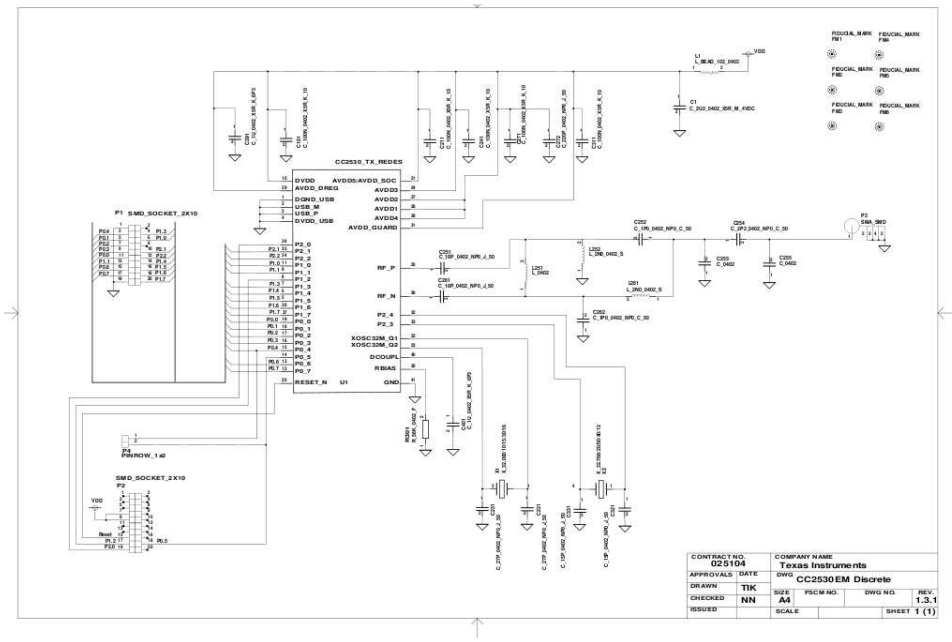

Figura 8.1: Implementación del CC2530 como CC2530EM propuesta por Texas Instruments

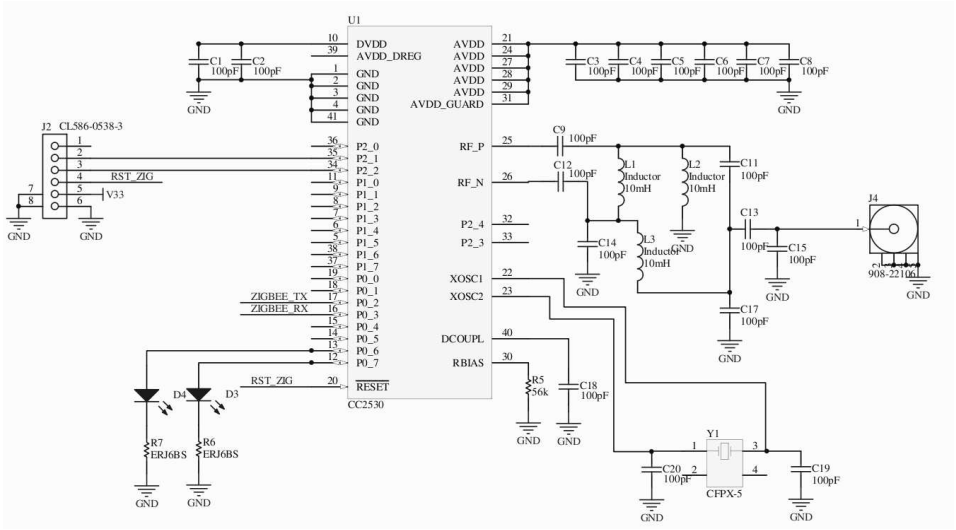

Figura 8.2: Implementación del CC2530 con un diseño similar al CC2530EM propuesto en el proyecto *weelrobot*

8.3. Adaptación del software

Para adaptar el software simplemente se modifica en el fichero **hal_board_cfg.h** en la sección **Drivers**. Por ejemplo, si no se quiere que las funciones que hacen uso del LCD actúen porque en la configuración de hardware definitiva no hay un LCD conectado como en la placa SmartRF05EB, se cambia la linea

Listing 8.1: Código en hal_board_cfg.h con la pantalla LCD habilitada

```
/* Set to TRUE enable LCD usage, FALSE disable it */
#ifndef HAL_LCD
#define HAL_LCD TRUE
#endif
```

por

Listing 8.2: Código en hal_board_cfg.h con la pantalla LCD deshabilitada

```
/* Set to TRUE enable LCD usage, FALSE disable it */
#ifndef HAL_LCD
#define HAL_LCD FALSE
#endif
```

en realidad se deben deshabilitar la mayoría de los periféricos, pero manteniendo el UART

Listing 8.3: Código en hal_board_cfg.h que habilita el UART

```
#define HAL_UART TRUE
```

Capítulo 9

Pasos siguientes

Tras el trabajo realizado, las lineas de mejora que se proponen son las siguientes:

- Diseño de una interfaz física estándar que permita la conexión a microcontroladores vía serie, sin necesidad de la placa SmartRF05EB.
- Diseño de una interfaz física estándar que permita la conexión a sistemas más complejos vía RS232 o RS323/USB, sin necesidad de la placa SmartRF05EB
- Implementación para el nodo **coordinador** del código en el *dongle* USB con el *SoC* CC2531 en lugar de con el CC2530.
- Definición de APIs de comandos más completas, una vez se haya completado la estandarización de todas las posibles órdenes.
- Mejora del código del puente *ZigBee*/UART para evitar la pérdida de paquetes.
- Implementación de rutinas inteligentes en el PC de control, posiblemente empleando el paradigma de máquina de estados.
- Gestión del control empleando herramientas más especializadas, como **JADE**.
- Implementación del CC2530 en proyectos del tipo *weelrobot*, de forma que se tengan dispositivos con comunicación *ZigBee* integrada en la misma placa.
- Verificación de la configuración USB/Serie en diferentes dispositivos, puesto que aún con dos dispositvos que implementen POSIX pueden requerir diferentes códigos de configuración, como se ha comprobado con el *Robotino*.
- Generalización del uso de la estructura de trama y de la tecnología *ZigBee* a todos los dispositivos y robots que requieran comunicación inalámbrica.

Capítulo 10

Conclusiones

- *ZigBee* es una excelente opción para la comunicación de robots y dispositivos móviles, que requieran un ancho de banda bajo y tengan importantes restricciones en cuanto a la energía que se puede dedicar a la comunicacion. *ZigBee* es adecuado en las comunicaciones de estado y de comandos.
- El dispositivo elegido, el *CC2530* permite, ya sea de forma individual montado en un empotrado siguiendo el diseño del *CC2530EM* o bien embarcado en la placa de desarrollo *SmartRF05EB* cumplir con las especificaciones de diseño coherentes con las características del protocolo *ZigBee*. Estas se pueden ver en su versión empotrada en el capítulo 8.
- La programación del *CC2530* se ha realizado en C bajo *IAR* y ha implicado la implementación de un puente *ZigBee*-UART bidireccional. De esta forma cuando un dispositivo envía vía UART una trama estandarizada, ésta contiene la información relativa al destino, el comando a ejecutar en destino y el formato de los datos. Al revés, cuando se recibe un mensaje *ZigBee* se desempaqueta la trama y se envía vía UART.
- Ha sido necesario diseñar una trama de 32 octetos que encapsula la información necesaria para su direccionamiento (dirección *ZigBee*, *end point* y *cluster*), en caso de que todos sean necesarios. También se han estandarizado los comandos. Es más fácil usar un estándar en comandos para una trama fija que trabajar por *clusters*, puesto que la programación vía *clusters* implica modificar la programación del *CC2530*, mientras que modificar el estándar de los comandos implica modificar el código en los PC y robots, normalmente más accesibles.
- En el lado del PC/Control se ha implementado en C una rutina de control que, basada en la librería de hilos de POSIX procesa mensajes entrantes y salientes, y permite generar adecuadamente una secuencia de órdenes y su envío cuya operativa más simple se indica en la sección 7.3. Las rutinas pueden ser más complejas, pero siempre dispondrán de, al menos tres hilos: uno para recepción, otro para envío y otro de toma de decisiones. La alternativa a la programación directa es emplear una herramienta tipo JADE, que exponga los servicios de una serie de robots o dispositivos a una rutina que procese las peticiones entrantes y mantenga un directorio de direcciones.
- El uso del *CC2530EM* se puede, gracias a la placa *SmartRF05EB* y a una conexión US-

B/Serie emplear como una alternativa de comunicación de, teóricamente[1], fácil conexión. Sin embargo, la integración recomendada es la descrita en el cápitulo 8 donde se puede, en un mismo empotrado, implementar el microcontrolador de control y el sistema de comunicación inalámbrico de bajo consumo.

[1]Realmente, aún compartiendo POSIX, *Robotino* y un PC con GNU-LINUX/Ubuntu no se configuran exactamente igual

Apéndice A

Software propuesto sobre CC2530 y SmartRF05

Listing A.1: Código GenericApp.c

```
1  /*********************************************************************
2   * INCLUDES
3   */
4  #include <string.h> // SCA
5
6
7  #include "OSAL.h"
8  #include "AF.h"
9  #include "ZDApp.h"
10 #include "ZDObject.h"
11 #include "ZDProfile.h"
12
13 #include "GenericApp.h"
14 #include "DebugTrace.h"
15
16 #if !defined( WIN32 )
17   #include "OnBoard.h"
18 #endif
19
20 /* HAL */
21 #include "hal_lcd.h"
22 #include "hal_led.h"
23 #include "hal_key.h"
24 #include "hal_uart.h"
25
26 /*********************************************************************
27  * MACROS
28  */
29
30 /*********************************************************************
31  * CONSTANTS
32  */
33
34 /*********************************************************************
35  * FUNCTIONS
36  */
37
```

```
38 void GenericApp_MessageMSGCBSCA(void);
39 void rellenarmensajeAenviar(afIncomingMSGPacket_t *pkt ,uint8 opcion);
40 void mostrarcomando(afIncomingMSGPacket_t *inpkt);
41 void GenericApp_SendTheMessageNEW(void);
42
43 /*********************************************************************
44  * TYPEDEFS
45  */
46
47 /*********************************************************************
48  * GLOBAL VARIABLES
49  */
50
51 uint8 mensajeAenviar32bytes[LONGITUD_TRAMA];
52 uint8 mensajeArecibir32bytes[LONGITUD_TRAMA];
53
54 zAddrType_t dstAddr;
55
56 // This list should be filled with Application specific Cluster IDs.
57 const cId_t GenericApp_ClusterList[GENERICAPP_MAX_CLUSTERS] =
58 {
59   GENERICAPP_CLUSTERID
60 };
61
62 const SimpleDescriptionFormat_t GenericApp_SimpleDesc =
63 {
64   GENERICAPP_ENDPOINT,              //  int Endpoint;
65   GENERICAPP_PROFID,                //  uint16 AppProfId[2];
66   GENERICAPP_DEVICEID,              //  uint16 AppDeviceId[2];
67   GENERICAPP_DEVICE_VERSION,        //  int   AppDevVer:4;
68   GENERICAPP_FLAGS,                 //  int   AppFlags:4;
69   GENERICAPP_MAX_CLUSTERS,          //  byte  AppNumInClusters;
70   (cId_t *)GenericApp_ClusterList,  //  byte *pAppInClusterList;
71   GENERICAPP_MAX_CLUSTERS,          //  byte  AppNumInClusters;
72   (cId_t *)GenericApp_ClusterList   //  byte *pAppInClusterList;
73 };
74
75 // This is the Endpoint/Interface description.  It is defined here, but
76 // filled-in in GenericApp_Init().  Another way to go would be to fill
77 // in the structure here and make it a "const" (in code space).  The
78 // way it's defined in this sample app it is define in RAM.
79 endPointDesc_t GenericApp_epDesc;
80
81 /*********************************************************************
82  * EXTERNAL VARIABLES
83  */
84 #if !defined( SERIAL_APP_PORT )
85 #define SERIAL_APP_PORT  0
86 #endif
87
88 #if !defined( SERIAL_APP_BAUD )
89 #define SERIAL_APP_BAUD  HAL_UART_BR_9600
90 //#define SERIAL_APP_BAUD  HAL_UART_BR_115200
91 #endif
92
93 // When the Rx buf space is less than this threshold, invoke the Rx callback.
94 #if !defined( SERIAL_APP_THRESH )
95 #define SERIAL_APP_THRESH  64
96 #endif
```

```
97
98 #if !defined( SERIAL_APP_RX_SZ )
99 #define SERIAL_APP_RX_SZ  128
100 #endif
101
102 #if !defined( SERIAL_APP_TX_SZ )
103 #define SERIAL_APP_TX_SZ  128
104 #endif
105
106 // Millisecs of idle time after a byte is received before invoking Rx callback.
107 #if !defined( SERIAL_APP_IDLE )
108 #define SERIAL_APP_IDLE  20
109 #endif
110
111 // Loopback Rx bytes to Tx for throughput testing.
112 #if !defined( SERIAL_APP_LOOPBACK )
113 #define SERIAL_APP_LOOPBACK  FALSE
114 #endif
115
116 // This is the max byte count per OTA message.
117 #if !defined( SERIAL_APP_TX_MAX )
118 #define SERIAL_APP_TX_MAX  80
119 #endif
120
121 #define SERIAL_APP_RSP_CNT  4
122
123 /*********************************************************************
124  * EXTERNAL FUNCTIONS
125  */
126
127 /*********************************************************************
128  * LOCAL VARIABLES
129  */
130 byte GenericApp_TaskID;   // Task ID for internal task/event processing
131                           // This variable will be received when
132                           // GenericApp_Init() is called.
133 devStates_t GenericApp_NwkState;
134
135
136 byte GenericApp_TransID;  // This is the unique message ID (counter)
137
138 afAddrType_t GenericApp_DstAddr;
139
140 /*  B - SERIAL COMMUNICATION  */
141 /*  SCA not sure if we'll use this one  */
142
143
144 uint8 SerialApp_TaskID;    // Task ID for internal task/event processing.
145 static uint8 SerialApp_MsgID;
146
147 static afAddrType_t SerialApp_TxAddr;
148 static uint8 SerialApp_TxSeq;
149 static uint8 SerialApp_TxBuf[SERIAL_APP_TX_MAX+1];
150 static uint8 SerialApp_TxLen = 0;
151
152 static afAddrType_t SerialApp_RxAddr;
153 static uint8 SerialApp_RxSeq;
154 static uint8 SerialApp_RspBuf[SERIAL_APP_RSP_CNT];
155 /*********************************************************************
```

```
156  * LOCAL FUNCTIONS
157  */
158 void GenericApp_ProcessZDOMsgs( zdoIncomingMsg_t *inMsg );
159 void GenericApp_HandleKeys( byte shift, byte keys );
160 void GenericApp_MessageMSGCB( afIncomingMSGPacket_t *pckt );
161 void GenericApp_SendTheMessage( void );
162 void enviarmensajeUART(afIncomingMSGPacket_t *MSGpkt);
163
164 /*  B - SERIAL COMMUNICATION  */
165
166 static void SerialApp_ProcessZDOMsgs( zdoIncomingMsg_t *inMsg );
167 static void SerialApp_HandleKeys( uint8 shift, uint8 keys );
168 static void SerialApp_ProcessMSGCmd( afIncomingMSGPacket_t *pkt );
169 static void SerialApp_Send(void);
170 static void SerialApp_Resp(void);
171 static void SerialApp_CallBack(uint8 port, uint8 event);
172
173 /*********************************************************************
174  * NETWORK LAYER CALLBACKS
175  */
176
177 /*********************************************************************
178  * PUBLIC FUNCTIONS
179  */
180
181 /*********************************************************************
182  * @fn      GenericApp_Init
183  *
184  * @brief   Initialization function for the Generic App Task.
185  *          This is called during initialization and should contain
186  *          any application specific initialization (ie. hardware
187  *          initialization/setup, table initialization, power up
188  *          notificaiton ... ).
189  *
190  * @param   task_id - the ID assigned by OSAL.  This ID should be
191  *                    used to send messages and set timers.
192  *
193  * @return  none
194  */
195 void GenericApp_Init( byte task_id )
196 {
197   GenericApp_TaskID = task_id;
198   GenericApp_NwkState = DEV_INIT;
199   GenericApp_TransID = 0;
200
201   // Device hardware initialization can be added here or in main() (Zmain.c).
202   // If the hardware is application specific - add it here.
203   // If the hardware is other parts of the device add it in main().
204
205   //  GenericApp_DstAddr.addrMode = (afAddrMode_t)AddrNotPresent;
206   GenericApp_DstAddr.addrMode = (afAddrMode_t)Addr16Bit;    //  Modified SCA
207   //  GenericApp_DstAddr.endPoint = 0;
208   GenericApp_DstAddr.endPoint  = GENERICAPP_ENDPOINT;
209   //  GenericApp_DstAddr.addr.shortAddr = 0;
210   GenericApp_DstAddr.addr.shortAddr = 0x0000;               //  Modified SCA
211
212   // Fill out the endpoint description.
213   GenericApp_epDesc.endPoint = GENERICAPP_ENDPOINT;
214   GenericApp_epDesc.task_id = &GenericApp_TaskID;
```

```
215   GenericApp_epDesc.simpleDesc
216             = (SimpleDescriptionFormat_t *)&GenericApp_SimpleDesc;
217   GenericApp_epDesc.latencyReq = noLatencyReqs;
218
219   // Register the endpoint description with the AF
220   afRegister( &GenericApp_epDesc );
221
222   // Register for all key events - This app will handle all key events
223   RegisterForKeys( GenericApp_TaskID );
224
225   // Update the display
226 #if defined ( LCD_SUPPORTED )
227     HalLcdWriteString( "GenericApp", HAL_LCD_LINE_1 );
228 #endif
229
230   ZDO_RegisterForZDOMsg( GenericApp_TaskID, End_Device_Bind_rsp );
231   ZDO_RegisterForZDOMsg( GenericApp_TaskID, Match_Desc_rsp );
232
233   //  added SCA
234    osal_start_timerEx( GenericApp_TaskID,
235                                GENERICAPP_SEND_MSG_EVT,
236                                GENERICAPP_SEND_MSG_TIMEOUT );
237
238    osal_start_timerEx( GenericApp_TaskID,
239                                GENERICAPP_FINDDEVICES,
240                                GENERICAPP_RETRYLINKING  );
241
242
243    /* B - SERIAL APP PART */
244
245   halUARTCfg_t uartConfig;
246
247   // SCA - maybe not needed
248   SerialApp_TaskID = task_id;
249   SerialApp_RxSeq = 0xC3;
250
251   //afRegister( (endPointDesc_t *)&SerialApp_epDesc );  //  Register the SECOND
          Endpoint
252                                                        //  as described
                                                               previously
253                                                        //  This EP has TWO
                                                               clusters
254
255   //  SCA already registered
256   //  RegisterForKeys( task_id );
257
258   uartConfig.configured           = TRUE;              // 2x30 don't care - see
          uart driver.
259   uartConfig.baudRate             = SERIAL_APP_BAUD;
260   uartConfig.flowControl          = TRUE;
261   uartConfig.flowControlThreshold = SERIAL_APP_THRESH; // 2x30 don't care - see
          uart driver.
262   uartConfig.rx.maxBufSize        = SERIAL_APP_RX_SZ;  // 2x30 don't care - see
          uart driver.
263   uartConfig.tx.maxBufSize        = SERIAL_APP_TX_SZ;  // 2x30 don't care - see
          uart driver.
264   uartConfig.idleTimeout          = SERIAL_APP_IDLE;   // 2x30 don't care - see
          uart driver.
```

```
265   uartConfig.intEnable            = TRUE;              // 2x30 don't care - see
          uart driver.
266   uartConfig.callBackFunc         = SerialApp_CallBack;
267   HalUARTOpen (SERIAL_APP_PORT, &uartConfig);
268
269   SerialApp_TxLen = 0;                                  //  lo inicializo aqu
270
271   /*********************************************************************
272    * @fn          ZDP_EndDeviceBindReq
273    *
274    * @brief       This builds and sends a End_Device_Bind_req message.
275    *              This function sends a unicast message.
276    *
277    * @param       dstAddr - destination address
278    * @param       LocalCoordinator - short address of local coordinator
279    * @param       epIntf - Endpoint/Interface of Simple Desc
280    * @param       ProfileID - Profile ID
281    *
282    *   The Input cluster list is the opposite of what you would think.
283    *   This is the output cluster list of this device
284    * @param       NumInClusters - number of input clusters
285    * @param       InClusterList - input cluster ID list
286    *
287    *   The Output cluster list is the opposite of what you would think.
288    *   This is the input cluster list of this device
289    * @param       NumOutClusters - number of output clusters
290    * @param       OutClusterList - output cluster ID list
291    *
292    * @param       SecurityEnable - Security Options
293    *
294    * @return      afStatus_t
295    */
296
297
298
299
300 }
301
302 /*********************************************************************
303    * @fn      GenericApp_ProcessEvent
304    *
305    * @brief   Generic Application Task event processor.  This function
306    *          is called to process all events for the task.  Events
307    *          include timers, messages and any other user defined events.
308    *
309    * @param   task_id  - The OSAL assigned task ID.
310    * @param   events - events to process.  This is a bit map and can
311    *                   contain more than one event.
312    *
313    * @return  none
314    */
315 UINT16 GenericApp_ProcessEvent( byte task_id, UINT16 events )
316 {
317   afIncomingMSGPacket_t *MSGpkt;
318   afDataConfirm_t *afDataConfirm;
319
320   // Data Confirmation message fields
321   byte sentEP;
322   ZStatus_t sentStatus;
```

```
323   byte sentTransID;       // This should match the value sent
324   (void)task_id;          // Intentionally unreferenced parameter
325
326   if ( events & SYS_EVENT_MSG )
327   {
328     MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( GenericApp_TaskID );
329     while ( MSGpkt )
330     {
331       switch ( MSGpkt->hdr.event )
332       {
333         case ZDO_CB_MSG:
334           GenericApp_ProcessZDOMsgs( (zdoIncomingMsg_t *)MSGpkt );
335           break;
336
337         case KEY_CHANGE:
338           //  GenericApp_HandleKeys( ((keyChange_t *)MSGpkt)->state, ((keyChange_t
                *)MSGpkt)->keys );
339           //  no se usan las teclas, que adem s en producci n estar n
                deshabilitadas
340           break;
341
342         case AF_DATA_CONFIRM_CMD:
343           // This message is received as a confirmation of a data packet sent.
344           // The status is of ZStatus_t type [defined in ZComDef.h]
345           // The message fields are defined in AF.h
346           afDataConfirm = (afDataConfirm_t *)MSGpkt;
347           sentEP = afDataConfirm->endpoint;
348           sentStatus = afDataConfirm->hdr.status;
349           sentTransID = afDataConfirm->transID;
350           (void)sentEP;
351           (void)sentTransID;
352
353           // Action taken when confirmation is received.
354           if ( sentStatus != ZSuccess )
355           {
356             // The data wasn't delivered -- Do something
357           }
358           break;
359
360         case AF_INCOMING_MSG_CMD:
361
362
363           //GenericApp_MessageMSGCB( MSGpkt );
364 #if HAL_LCD == TRUE
365           mostrarcomando(MSGpkt);
366 #endif
367
368           //  rellenarmensajeAenviar(MSGpkt ,1);      //  No se devuelve mensaje
                OTA, al ser un
369                                                     //  mero puente se delega en
                                                            el dispositivo
370                                                     //  conectado v a serie la
                                                            decisi n de qu
371                                                     //  hacer
372           enviarmensajeUART(MSGpkt);                //  envia mensaje por uart
373           break;
374
375         case ZDO_STATE_CHANGE:
376           GenericApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
```

```
377
378
379           //  SCA modification
380
381 #if MENSAJE_PERIODICO == TRUE          //  definido en GenericApp.h
382           if(GenericApp_NwkState == DEV_END_DEVICE)
383           {
384               //SCA send every 5 sec ONLY if End Device
385               osal_start_timerEx( GenericApp_TaskID,
386                                   GENERICAPP_SEND_MSG_EVT,
387                                   GENERICAPP_SEND_MSG_TIMEOUT );
388           }
389 #endif
390           break;
391
392         default:
393           break;
394       }
395
396       // Release the memory
397       osal_msg_deallocate( (uint8 *)MSGpkt );
398
399       // Next
400       MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( GenericApp_TaskID );
401     }
402
403     // return unprocessed events
404     return (events ^ SYS_EVENT_MSG);
405   }
406
407   // Send a message out - This event is generated by a timer
408   //  (setup in GenericApp_Init()).
409   if ( events & GENERICAPP_SEND_MSG_EVT )
410   {
411
412 #ifdef  MENSAJE_PERIODICO == TRUE
413     GenericApp_SendTheMessageNEW();
414 #endif
415
416     return (events ^ GENERICAPP_SEND_MSG_EVT);
417   }
418
419
420   if ( events & GENERICAPP_FINDDEVICES )
421   {
422
423       return (events ^ GENERICAPP_FINDDEVICES );
424
425     }
426
427
428   //}
429
430   // Discard unknown events
431   return 0;
432 }
433
434 /*********************************************************************
435  * Event Generation Functions
```

```
436   */
437
438 /*********************************************************************
439   * @fn      GenericApp_ProcessZDOMsgs()
440   *
441   * @brief   Process response messages
442   *
443   * @param   none
444   *
445   * @return  none
446   */
447 void GenericApp_ProcessZDOMsgs( zdoIncomingMsg_t *inMsg )
448 {
449   switch ( inMsg->clusterID )
450   {
451     case End_Device_Bind_rsp:
452       if ( ZDO_ParseBindRsp( inMsg ) == ZSuccess )
453       {
454         // Light LED
455         HalLedSet( HAL_LED_4, HAL_LED_MODE_ON );
456       }
457 #if defined( BLINK_LEDS )
458       else
459       {
460         // Flash LED to show failure
461         HalLedSet ( HAL_LED_4, HAL_LED_MODE_FLASH );
462       }
463 #endif
464       break;
465
466     case Match_Desc_rsp:
467       {
468         ZDO_ActiveEndpointRsp_t *pRsp = ZDO_ParseEPListRsp( inMsg );
469         if ( pRsp )
470         {
471           if ( pRsp->status == ZSuccess && pRsp->cnt )
472           {
473             GenericApp_DstAddr.addrMode = (afAddrMode_t)Addr16Bit;
474             GenericApp_DstAddr.addr.shortAddr = pRsp->nwkAddr;
475             // Take the first endpoint, Can be changed to search through endpoints
476             GenericApp_DstAddr.endPoint = pRsp->epList[0];
477
478             // Light LED
479             HalLedSet( HAL_LED_4, HAL_LED_MODE_ON );
480           }
481           osal_mem_free( pRsp );
482         }
483       }
484       break;
485   }
486 }
487
488 /*********************************************************************
489   * @fn      GenericApp_HandleKeys
490   *
491   * @brief   Handles all key events for this device.
492   *
493   * @param   shift - true if in shift/alt.
494   * @param   keys - bit field for key events. Valid entries:
```

```
495  *                  HAL_KEY_SW_4
496  *                  HAL_KEY_SW_3
497  *                  HAL_KEY_SW_2
498  *                  HAL_KEY_SW_1
499  *
500  * @return  none
501  */
502 void GenericApp_HandleKeys( byte shift, byte keys )
503 {
504   // zAddrType_t dstAddr;
505
506   // Shift is used to make each button/switch dual purpose.
507   if ( shift )
508   {
509     if ( keys & HAL_KEY_SW_1 )
510     {
511         HalLcdWriteString( "GenericApp", HAL_LCD_LINE_1 );
512     }
513     if ( keys & HAL_KEY_SW_2 )
514     {
515     }
516     if ( keys & HAL_KEY_SW_3 )
517     {
518     }
519     if ( keys & HAL_KEY_SW_4 )
520     {
521     }
522     if ( keys & HAL_KEY_SW_6 )
523     {
524       HalLcdWriteString( "BUTTON_2_UP", HAL_LCD_LINE_1 );
525     }
526
527
528
529   }
530   else
531   {
532     if ( keys & HAL_KEY_SW_1 )
533     {
534       HalLcdWriteString( "KEY_1_UP", HAL_LCD_LINE_1 );
535       // Since SW1 isn't used for anything else in this application...
536 #if defined( SWITCH1_BIND )
537       // we can use SW1 to simulate SW2 for devices that only have one switch,
538       keys |= HAL_KEY_SW_2;
539 #elif defined( SWITCH1_MATCH )
540       // or use SW1 to simulate SW4 for devices that only have one switch
541       keys |= HAL_KEY_SW_4;
542 #endif
543     }
544
545     if ( keys & HAL_KEY_SW_2 )
546     {
547       HalLcdWriteString( "Binding...", HAL_LCD_LINE_1 );
548       HalLedSet ( HAL_LED_4, HAL_LED_MODE_OFF );
549
550       // Initiate an End Device Bind Request for the mandatory endpoint
551       dstAddr.addrMode = Addr16Bit;
552       dstAddr.addr.shortAddr = 0x0000; // Coordinator
553       ZDP_EndDeviceBindReq( &dstAddr, NLME_GetShortAddr(),
```

```
554                                 GenericApp_epDesc.endPoint,
555                                 GENERICAPP_PROFID,
556                                 GENERICAPP_MAX_CLUSTERS, (cId_t *)
                                        GenericApp_ClusterList,
557                                 GENERICAPP_MAX_CLUSTERS, (cId_t *)
                                        GenericApp_ClusterList,
558                                 FALSE );
559     }
560
561     if ( keys & HAL_KEY_SW_3 )
562     {
563     }
564
565     if ( keys & HAL_KEY_SW_4 )
566     {
567       HalLedSet ( HAL_LED_4, HAL_LED_MODE_OFF );
568       // Initiate a Match Description Request (Service Discovery)
569       dstAddr.addrMode = AddrBroadcast;
570       dstAddr.addr.shortAddr = NWK_BROADCAST_SHORTADDR;
571       ZDP_MatchDescReq( &dstAddr, NWK_BROADCAST_SHORTADDR,
572                         GENERICAPP_PROFID,
573                         GENERICAPP_MAX_CLUSTERS, (cId_t *)GenericApp_ClusterList,
574                         GENERICAPP_MAX_CLUSTERS, (cId_t *)GenericApp_ClusterList,
575                         FALSE );
576     }
577
578     if ( keys & HAL_KEY_SW_5 )
579     {
580       HalLcdWriteString( "J_PRESS_DOWN", HAL_LCD_LINE_1 );
581       HalLedSet ( HAL_LED_1, HAL_LED_MODE_ON );
582     }
583
584     if ( keys & HAL_KEY_SW_6 )
585     {
586       HalLcdWriteString( "BUTTON_2_UP", HAL_LCD_LINE_1 );
587       HalLedSet ( HAL_LED_2, HAL_LED_MODE_ON );
588     }
589
590   }
591 }
592
593 /*********************************************************************
594  * LOCAL FUNCTIONS
595  */
596
597 /*********************************************************************
598  * @fn      GenericApp_MessageMSGCB
599  *
600  * @brief   Data message processor callback.  This function processes
601  *          any incoming data - probably from other devices.  So, based
602  *          on cluster ID, perform the intended action.
603  *
604  * @param   none
605  *
606  * @return  none
607  */
608 void GenericApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
609 {
610   switch ( pkt->clusterId )
```

```
611   {
612     case GENERICAPP_CLUSTERID:
613     // "the" message
614 #if defined( LCD_SUPPORTED )
615       //  HalLcdWriteScreen( (char*)pkt->cmd.Data, "rcvd" ); //SCA (line1, line2)
616
617 /*************************************************
618  * @fn      HalLcdWriteStringValue
619  *
620  * @brief   Write a string followed by a value to the LCD
621  *
622  * @param   title  - Title that will be displayed before the value
623  *          value  - value
624  *          format - redix
625  *          line   - line number
626  *
627  * @return  None
628  ************************************************************************/
629       HalLcdWriteString((char*)pkt->cmd.Data , 1 );
630       HalLcdWriteStringValue("Origen: ", pkt->srcAddr.addr.shortAddr, 16, 2 );
631
632       /*  ONLY if this is the coordinator   */
633       char theMessageDataResponse[] = "COORD_AWARE!";
634       if ( (GenericApp_NwkState == DEV_ZB_COORD) )
635       {
636             GenericApp_DstAddr.addr.shortAddr = pkt->srcAddr.addr.shortAddr;    //
                    SCA change the destination address
637             AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc,
638                       GENERICAPP_CLUSTERID,
639                       (byte)osal_strlen( theMessageDataResponse ) + 1,
640                       (byte *)&theMessageDataResponse,
641                       &GenericApp_TransID,
642                       AF_DISCV_ROUTE, AF_DEFAULT_RADIUS );
643
644
645       }
646
647 #elif defined( WIN32 )
648       WPRINTSTR( pkt->cmd.Data );
649 #endif
650       break;
651   }
652 }
653
654 /*************************************************
655  * @fn      GenericApp_SendTheMessage
656  *
657  * @brief   Send "the" message.
658  *
659  * @param   none
660  *
661  * @return  none
662  */
663 void GenericApp_SendTheMessage( void )
664 {
665   char theMessageData[] = "ENDDEV_HERE!";
666
667   /*************************************************
668  * @fn      AF_DataRequest
```

```
669  *
670  * @brief   Common functionality for invoking APSDE_DataReq() for both
671  *          SendMulti and MSG-Send.
672  *
673  * input parameters
674  *
675  * @param  *dstAddr - Full ZB destination address: Nwk Addr + End Point.
676  * @param  *srcEP - Origination (i.e. respond to or ack to) End Point Descr.
677  * @param   cID - A valid cluster ID as specified by the Profile.
678  * @param   len - Number of bytes of data pointed to by next param.
679  * @param  *buf - A pointer to the data bytes to send.
680  * @param  *transID - A pointer to a byte which can be modified and which will
681  *                    be used as the transaction sequence number of the msg.
682  * @param   options - Valid bit mask of Tx options.
683  * @param   radius - Normally set to AF_DEFAULT_RADIUS.
684  *
685  * output parameters
686  *
687  * @param  *transID - Incremented by one if the return value is success.
688  *
689  * @return  afStatus_t - See previous definition of afStatus_... types.
690  */
691   if ( (GenericApp_NwkState == DEV_END_DEVICE) )
692   {
693     HalLcdWriteString( "Mensaje Enviado", HAL_LCD_LINE_2 );
694     if ( AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc,
695                          GENERICAPP_CLUSTERID,
696                          (byte)osal_strlen( theMessageData ) + 1,
697                          (byte *)&theMessageData,
698                          &GenericApp_TransID,
699                          AF_DISCV_ROUTE, AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
700     {
701       // Successfully requested to be sent.
702     }
703     else
704     {
705       // Error occurred in request to send.
706     }
707   }
708 }
709
710 /*********************************************
711  * @fn      SerialApp_CallBack
712  *
713  * @brief   Send data OTA.
714  *
715  * @param   port - UART port.
716  * @param   event - the UART port event flag.
717  *
718  * @return  none
719  */
720 static void SerialApp_CallBack(uint8 port, uint8 event)
721 {
722
723
724   (void)port;
725   /*********************************************
726  * @fn      HalUARTRead
727  *
```

```
728  * @brief   Read a buffer from the UART
729  *
730  * @param   port - USART module designation
731  *          buf  - valid data buffer at least 'len' bytes in size
732  *          len  - max length number of bytes to copy to 'buf'
733  *
734  * @return  length of buffer that was read
735  *****************************************************************************/
736 //  uint16 HalUARTRead(uint8 port, uint8 *buf, uint16 len)
737
738   //  start writing only IF SerialApp_TxLen = 0 and the incoming byte is 0
739
740
741    //SerialApp_TxLen += HalUARTRead(SERIAL_APP_PORT, SerialApp_TxBuf +
          SerialApp_TxLen,1);
742    //HalUARTRead(SERIAL_APP_PORT, SerialApp_TxBuf + SerialApp_TxLen,1);
743    //SerialApp_TxLen++;
744
745    HalUARTRead(SERIAL_APP_PORT, SerialApp_TxBuf, LONGITUD_TRAMA);
746    HalLedSet (HAL_LED_4, HAL_LED_MODE_TOGGLE); //  Para verificar que se ha
          recibido un byte desde el
747                                                //  equipo, en este caso el
                                                      ordenador
748    // significa que ya podemos enviar
749    //if((SerialApp_TxLen == (LONGITUD_TRAMA - 1)) && (SerialApp_TxBuf[0] == 0xFF)
          && (SerialApp_TxBuf[31] == '\0'))
750    if((SerialApp_TxBuf[0] == 0xFF) && (SerialApp_TxBuf[31] == '\0'))
751    {
752           //  fill in the right buffer
753           for(int i = 0;i < LONGITUD_TRAMA;i++){mensajeAenviar32bytes[i] =
                  SerialApp_TxBuf[i];}
754
755           //  clean the buffer to prevent echoes
756
757           for(int i = 0; i < LONGITUD_TRAMA; i++){ SerialApp_TxBuf[i] = 0xAA ;}
758
759           // send the mensajeAenviar32bytes
760
761           GenericApp_MessageMSGCBSCA();
762
763           //  counter to zero
764
765           HalLedSet (HAL_LED_3, HAL_LED_MODE_TOGGLE);
766    }
767
768
769
770
771 }
772
773
774
775 //  Esta funci n no se usa en esta implementaci n
776 void rellenarmensajeAenviar(afIncomingMSGPacket_t *pkt ,uint8 opcion)
777 {
778
779   /* typedef struct{
780
781     //  0     0xFF
```

```
782
783     //  1-2   origin node address
784     uint8* originnode2bytes;
785     //  3     EP origin
786     uint8* endpointorigin1byte;
787     //  4-5   destination node address
788     uint8* destinationnode2bytes;
789     //  6     EP destination
790     uint8* endpointdestination1byte;
791     //  7     Destination cluster
792     uint8*  destinationcluster1byte;
793     //  8-9 16-bit command
794     uint8* command2bytes;
795     //  10-31 21 bytes of data
796     uint8*  bytesofdata21;
797
798   } framedata_t; */
799
800   switch(opcion)
801   {
802       case 1:
803             //  inicio de la trama/frame
804             mensajeAenviar32bytes[0] = 0xFF;
805             mensajeAenviar32bytes[1] = (uint8)(NLME_GetShortAddr());
                                          //  LSB
806             mensajeAenviar32bytes[2] = (uint8)(NLME_GetShortAddr() >> 8);
                                       //  MSB
807             //  Origin EP
808             mensajeAenviar32bytes[3] = GENERICAPP_ENDPOINT;
809
810             //  destination node address LSB
811             mensajeAenviar32bytes[4] =(uint8) (pkt->srcAddr.addr.shortAddr);
                                     //  ESTO LA HACE BIEN
812             //  destination node address MSB
813             mensajeAenviar32bytes[5] =(uint8) ((pkt->srcAddr.addr.shortAddr) >> 8)
                    ;   //  ESTO LO HACE BIEN
814
815             //  Destination EP
816             mensajeAenviar32bytes[6] = GENERICAPP_ENDPOINT;
817
818             //  Destination cluster (in this case I dont use clusters)
819             mensajeAenviar32bytes[7] = 0;     //  not relevant in this application
820
821             //if ( (GenericApp_NwkState == DEV_ZB_COORD) )
822             //{  //  command LSB
823             //  mensajeAenviar32bytes[8] = 0x51;  //  como prueba
824               //  command MSB
825             //  mensajeAenviar32bytes[9] = 0x53;  //  como prueba
826             //  }
827             if ( (GenericApp_NwkState == DEV_END_DEVICE) )
828             {
829
830               mensajeAenviar32bytes[8] =  (uint8) 0x7B;
831               mensajeAenviar32bytes[9] =  (uint8) 0x7D;
832             }
833             //  data fields (10 - 31) not used now
834
835         break;
836
```

```
837         case 2:
838
839           break;
840
841         case 3:
842
843           break;
844
845         default:
846
847           break;
848
849     }
850           GenericApp_MessageMSGCBSCA();
851
852 }
853
854
855 void GenericApp_MessageMSGCBSCA(void)
856 {
857
858 /*********************************************
859  * @fn      HalLcdWriteStringValue
860  *
861  * @brief   Write a string followed by a value to the LCD
862  *
863  * @param   title  - Title that will be displayed before the value
864  *          value  - value
865  *          format - redix
866  *          line   - line number
867  *
868  * @return  None
869  ********************************************************/
870       //  HalLcdWriteString((char*)pkt->cmd.Data , 1 );
871       //  HalLcdWriteStringValue("Origen: ", pkt->srcAddr.addr.shortAddr, 16, 2 );
872
873       /*********************************************
874  * @fn      AF_DataRequest
875  *
876  * @brief   Common functionality for invoking APSDE_DataReq() for both
877  *          SendMulti and MSG-Send.
878  *
879  * input parameters
880  *
881  * @param  *dstAddr - Full ZB destination address: Nwk Addr + End Point.
882  * @param  *srcEP - Origination (i.e. respond to or ack to) End Point Descr.
883  * @param   cID - A valid cluster ID as specified by the Profile.
884  * @param   len - Number of bytes of data pointed to by next param.
885  * @param  *buf - A pointer to the data bytes to send.
886  * @param  *transID - A pointer to a byte which can be modified and which will
887  *                    be used as the transaction sequence number of the msg.
888  * @param   options - Valid bit mask of Tx options.
889  * @param   radius - Normally set to AF_DEFAULT_RADIUS.
890  *
891  * output parameters
892  *
893  * @param  *transID - Incremented by one if the return value is success.
894  *
895  * @return  afStatus_t - See previous definition of afStatus_... types.
```

```
896  */
897
898 /* typedef struct{
899
900     //  0      0xFF
901
902     //  1-2    origin node address
903     uint8* originnode2bytes;
904     //  3      EP origin
905     uint8* endpointorigin1byte;
906     //  4-5    destination node address
907     uint8* destinationnode2bytes;
908     //  6      EP destination
909     uint8* endpointdestination1byte;
910     //  7      Destination cluster
911     uint8*  destinationcluster1byte;
912     //  8-9 16-bit command
913     uint8* command2bytes;
914     //  10-31 21 bytes of data
915     uint8*  bytesofdata21;
916
917   } framedata_t; */
918
919
920       /*  ONLY if this is the coordinator    */
921       // char theMessageDataResponse[] = "COORD_AWARE!";
922       //if ( (GenericApp_NwkState == DEV_ZB_COORD) )
923       //{
924             //  GenericApp_DstAddr.addr.shortAddr = pkt->srcAddr.addr.shortAddr;
                          //  SCA change the destination address
925
926             GenericApp_DstAddr.addr.shortAddr = (uint16)(mensajeAenviar32bytes[4]
                  & 0x00ff) | ((mensajeAenviar32bytes[5] << 8) & 0xff00 );        //
                   ESTO TB LO HACE BIEN
927             //  A donde env a ...
928             HalLcdWriteStringValue("Destino: ", GenericApp_DstAddr.addr.shortAddr
                  ,16,3);  //  ESTO TB LO HACE BIEN
929             //  adem s debe rellenar los campos de ORIGEN (direcci n de origen y
                   EP)
930
931             mensajeAenviar32bytes[1] = (uint8)(NLME_GetShortAddr());
                                  //  LSB
932             mensajeAenviar32bytes[2] = (uint8)(NLME_GetShortAddr() >> 8);
                               //  MSB
933             //  Origin EP
934             mensajeAenviar32bytes[3] = GENERICAPP_ENDPOINT;
                                         //  EP
935
936
937             //  Orden de envio ...
938             AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc,
939                       GENERICAPP_CLUSTERID,
940                       LONGITUD_TRAMA,
941                       (byte *)&mensajeAenviar32bytes,
942                       &GenericApp_TransID,
943                       AF_DISCV_ROUTE, AF_DEFAULT_RADIUS );
944
945            // limpiamos el buffer de envio para evitar ecos
946
```

```
947             for(int i = 0; i < LONGITUD_TRAMA; i++){mensajeAenviar32bytes[i] = 0
                    xAA;}
948
949         //}
950 }
951
952
953
954 void mostrarcomando(afIncomingMSGPacket_t *inpkt)
955 {
956
957     //  uint8 message[32];
958     //  message = &(inpkt->cmd.Data[4]);
959
960     //uint16 lectura = (uint16)((inpkt->cmd.Data[4]) & 0x00ff) | (((inpkt->cmd.
            Data[5]) << 8) & 0xff00 );
961   if ( (GenericApp_NwkState == DEV_END_DEVICE) ){ HalLedSet (HAL_LED_2,
          HAL_LED_MODE_TOGGLE);}
962   else{HalLedSet (HAL_LED_1, HAL_LED_MODE_TOGGLE);}
963
964
965     HalLcdWriteStringValue("Origen: ", (uint16)((inpkt->cmd.Data[4]) & 0x00ff) |
            (((inpkt->cmd.Data[5]) << 8) & 0xff00 ), 16,2);
966     HalLcdWriteStringValue("Comando: ", (uint16)((inpkt->cmd.Data[8]) & 0x00ff) |
            (((inpkt->cmd.Data[9]) << 8) & 0xff00 ), 16,1);
967
968
969     //  HalLcdWriteStringValue("Length: ", (uint16)inpkt->cmd.DataLength, 10, 2);
                //  ESTE FUNCIONA BIEN
970
971 }
972
973
974 void GenericApp_SendTheMessageNEW()
975 {
976
977  //   Inicio de la trama/frame
978  //   Este mensaje es de identificaci n
979             mensajeAenviar32bytes[0] = 0xFF;
980
981             mensajeAenviar32bytes[1] = (uint8)(NLME_GetShortAddr());
                                    //  LSB
982             mensajeAenviar32bytes[2] = (uint8)(NLME_GetShortAddr() >> 8);
                                //  MSB
983             //  Origin EP
984             mensajeAenviar32bytes[3] = GENERICAPP_ENDPOINT;
985
986             //  destination node address LSB
987             mensajeAenviar32bytes[4] =(uint8) 0x00;          //   ENVIAR  CADA 30
                    s a Coordinator
988             //  destination node address MSB
989             mensajeAenviar32bytes[5] =(uint8) 0x00;           //  ENVIAR  CADA 30
                    s a Coordinator
990
991             //  Destination EP
992             mensajeAenviar32bytes[6] = GENERICAPP_ENDPOINT;
993
994             //  Destination cluster (in this case I dont use clusters)
995             mensajeAenviar32bytes[7] = 0;     //  not relevant in this application
```

```
996
997
998             //  command LSB
999             mensajeAenviar32bytes[8] = 0x00;     //  version definitiva
1000            //  command MSB
1001            mensajeAenviar32bytes[9] = 0x00;      //  version definitiva
1002
1003
1004            //  data fields (10 - 31)
1005            mensajeAenviar32bytes[10] = 'M' ;
1006            mensajeAenviar32bytes[11] = 'e';
1007            mensajeAenviar32bytes[12] = 'n';
1008            mensajeAenviar32bytes[13] = 's';
1009            mensajeAenviar32bytes[14] = 'a';
1010            mensajeAenviar32bytes[15] = 'j';
1011            mensajeAenviar32bytes[16] = 'e';
1012            mensajeAenviar32bytes[17] = ' ';
1013            mensajeAenviar32bytes[18] = 'p';
1014            mensajeAenviar32bytes[19] = 'e';
1015            mensajeAenviar32bytes[20] = 'r';
1016            mensajeAenviar32bytes[21] = 'i';
1017            mensajeAenviar32bytes[22] = 'o';
1018            mensajeAenviar32bytes[23] = 'd';
1019            mensajeAenviar32bytes[24] = 'i';
1020            mensajeAenviar32bytes[25] = 'c';
1021            mensajeAenviar32bytes[26] = 'o';
1022            mensajeAenviar32bytes[27] = '.';
1023            mensajeAenviar32bytes[28] = '.';
1024            mensajeAenviar32bytes[29] = '.';
1025            mensajeAenviar32bytes[30] = '.';
1026
1027
1028
1029            mensajeAenviar32bytes[31] = '\0';
1030
1031 #if MENSAJE_PERIODICO == TRUE
1032            //  Enviar solo si EndDevice
1033             if ( (GenericApp_NwkState == DEV_END_DEVICE) )
1034             {
1035                     GenericApp_MessageMSGCBSCA();
1036                      //   si se env a se reenv a otra vez, de esta forma
1037                      //   se tiene un mensaje que se env a desde este
                             ENDDEVICE
1038                      //   (en el caso de que lo sea) al COORDINATOR
1039                     osal_start_timerEx( GenericApp_TaskID,
1040                             GENERICAPP_SEND_MSG_EVT,
1041                             GENERICAPP_SEND_MSG_TIMEOUT );
1042             }
1043
1044 #endif
1045
1046 }
1047
1048
1049
1050 void enviarmensajeUART(afIncomingMSGPacket_t *pkt)
1051 {
1052         if(pkt->cmd.DataLength == LONGITUD_TRAMA)
1053         {
```

```
1054                HalLedSet (HAL_LED_3, HAL_LED_MODE_TOGGLE);
1055                HalUARTWrite(SERIAL_APP_PORT, pkt->cmd.Data, LONGITUD_TRAMA);
1056        }
1057 }
```

Listing A.2: Código GenericApp.h

```
1
2  #ifndef GENERICAPP_H
3  #define GENERICAPP_H
4
5  #ifdef __cplusplus
6  extern "C"
7  {
8  #endif
9
10 /*********************************************************************
11  * INCLUDES
12  */
13 #include "ZComDef.h"
14
15 /*********************************************************************
16  * CONSTANTS
17  */
18
19 // These constants are only for example and should be changed to the
20 // device's needs
21 #define GENERICAPP_ENDPOINT            10
22
23 #define GENERICAPP_PROFID              0x0F04
24 #define GENERICAPP_DEVICEID            0x0001
25 #define GENERICAPP_DEVICE_VERSION      0
26 #define GENERICAPP_FLAGS               0
27
28 #define GENERICAPP_MAX_CLUSTERS        1
29 #define GENERICAPP_CLUSTERID           1
30
31 // Send Message Timeout
32 #define GENERICAPP_SEND_MSG_TIMEOUT    30000     //  Every 30 seconds
33 #define GENERICAPP_RETRYLINKING        10000     //  Every after 10 seconds
34
35 // Application Events (OSAL) - These are bit weighted definitions.
36 #define GENERICAPP_SEND_MSG_EVT        0x0001
37 #define GENERICAPP_FINDDEVICES         0x0004    //  Every second
38
39 /* SCA para integrar UART */
40
41 //#define SERIALAPP_MAX_CLUSTERS       2
42 #define SERIALAPP_CLUSTERID1          2  //SCA used to be 1
43 #define SERIALAPP_CLUSTERID2          3  //SCA used to be 2
44
45
46 //  SCA - changed because shared values with GENERICAPP_SEND_MSG_VT
47 #define SERIALAPP_SEND_EVT   0x0002
48 #define SERIALAPP_RESP_EVT   0x0003
49
50 //  SCA
51 #define LONGITUD_TRAMA   32
52
```

```
#define MENSAJE_PERIODICO    FALSE //  si FALSE no se envia mensaje periodico
                                   //  si TRUE se env a mensaje periodico

/*********************************************************************
 * MACROS
 */

/*********************************************************************
 * TYPEDEFS
 */

  typedef struct{

    //  0      0xFF

    //  1-2    origin node address
    uint8* originnode2bytes;
    //  3      EP origin
    uint8* endpointorigin1byte;
    //  4-5    destination node address
    uint8* destinationnode2bytes;
    //  6      EP destination
    uint8* endpointdestination1byte;
    //  7      Destination cluster
    uint8*  destinationcluster1byte;
    //  8-9 16-bit command
    uint8* command2bytes;
    //  10-31 21 bytes of data
    uint8*  bytesofdata21;

  } framedata_t;



/*********************************************************************
 * FUNCTIONS
 */

/*
 * Task Initialization for the Generic Application
 */
extern void GenericApp_Init( byte task_id );

/*
 * Task Event Processor for the Generic Application
 */
extern UINT16 GenericApp_ProcessEvent( byte task_id, UINT16 events );

/*********************************************************************
*********************************************************************/

#ifdef __cplusplus
}
#endif

#endif /* GENERICAPP_H */
```

Listing A.3: Código GenericApp.h

```
1
2  #include "ZComDef.h"
3  #include "hal_drivers.h"
4  #include "OSAL.h"
5  #include "OSAL_Tasks.h"
6
7  #if defined ( MT_TASK )
8    #include "MT.h"
9    #include "MT_TASK.h"
10 #endif
11
12 #include "nwk.h"
13 #include "APS.h"
14 #include "ZDApp.h"
15 #if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
16   #include "ZDNwkMgr.h"
17 #endif
18 #if defined ( ZIGBEE_FRAGMENTATION )
19   #include "aps_frag.h"
20 #endif
21
22 #include "GenericApp.h"
23
24 /*********************************************************************
25  * GLOBAL VARIABLES
26  */
27
28
29
30
31 // The order in this table must be identical to the task initialization calls
       below in osalInitTask.
32 const pTaskEventHandlerFn tasksArr[] = {
33   macEventLoop,
34   nwk_event_loop,
35   Hal_ProcessEvent,
36 #if defined( MT_TASK )
37   MT_ProcessEvent,
38 #endif
39   APS_event_loop,
40 #if defined ( ZIGBEE_FRAGMENTATION )
41   APSF_ProcessEvent,
42 #endif
43   ZDApp_event_loop,
44 #if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
45   ZDNwkMgr_event_loop,
46 #endif
47   GenericApp_ProcessEvent
48 };
49
50 const uint8 tasksCnt = sizeof( tasksArr ) / sizeof( tasksArr[0] );
51 uint16 *tasksEvents;
52
53 /*********************************************************************
54  * FUNCTIONS
55  *********************************************************************/
56
57 /*********************************************************************
58  * @fn      osalInitTasks
```

```
59  *
60  * @brief   This function invokes the initialization function for each task.
61  *
62  * @param   void
63  *
64  * @return  none
65  */
66 void osalInitTasks( void )
67 {
68   uint8 taskID = 0;
69
70   tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
71   osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));
72
73   macTaskInit( taskID++ );
74   nwk_init( taskID++ );
75   Hal_Init( taskID++ );
76 #if defined( MT_TASK )
77   MT_TaskInit( taskID++ );
78 #endif
79   APS_Init( taskID++ );
80 #if defined ( ZIGBEE_FRAGMENTATION )
81   APSF_Init( taskID++ );
82 #endif
83   ZDApp_Init( taskID++ );
84 #if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
85   ZDNwkMgr_Init( taskID++ );
86 #endif
87   GenericApp_Init( taskID );
88 }
```

Listing A.4: f8wconfig.cfg

```
1
2 /* Enable ZigBee-Pro */
3 -DZIGBEEPRO
4
5 /* Set to 0 for no security, otherwise non-0 */
6 -DSECURE=0
7 -DZG_SECURE_DYNAMIC=0
8
9 /* Enable the Reflector */
10 -DREFLECTOR
11
12 /* Default channel is Channel 11 - 0x0B */
13 // Channels are defined in the following:
14 //          0      : 868 MHz      0x00000001
15 //          1 - 10 : 915 MHz      0x000007FE
16 //         11 - 26 : 2.4 GHz      0x07FFF800
17 //
18 //-DMAX_CHANNELS_868MHZ      0x00000001
19 //-DMAX_CHANNELS_915MHZ      0x000007FE
20 //-DMAX_CHANNELS_24GHZ       0x07FFF800
21 //-DDEFAULT_CHANLIST=0x04000000  // 26 - 0x1A
22 //-DDEFAULT_CHANLIST=0x02000000  // 25 - 0x19
23 //-DDEFAULT_CHANLIST=0x01000000  // 24 - 0x18
24 //-DDEFAULT_CHANLIST=0x00800000  // 23 - 0x17
25 //-DDEFAULT_CHANLIST=0x00400000  // 22 - 0x16
26 //-DDEFAULT_CHANLIST=0x00200000  // 21 - 0x15
```

```
27 //-DDEFAULT_CHANLIST=0x00100000  // 20 - 0x14
28 //-DDEFAULT_CHANLIST=0x00080000  // 19 - 0x13
29 //-DDEFAULT_CHANLIST=0x00040000  // 18 - 0x12
30 //-DDEFAULT_CHANLIST=0x00020000  // 17 - 0x11
31 //-DDEFAULT_CHANLIST=0x00010000  // 16 - 0x10
32 //-DDEFAULT_CHANLIST=0x00008000  // 15 - 0x0F
33 //-DDEFAULT_CHANLIST=0x00004000  // 14 - 0x0E
34 //-DDEFAULT_CHANLIST=0x00002000  // 13 - 0x0D
35 //-DDEFAULT_CHANLIST=0x00001000  // 12 - 0x0C
36 -DDEFAULT_CHANLIST=0x00000800  // 11 - 0x0B
37
38 /* Define the default PAN ID.
39  *
40  * Setting this to a value other than 0xFFFF causes
41  * ZDO_COORD to use this value as its PAN ID and
42  * Routers and end devices to join PAN with this ID
43  */
44  /* SCA I have decided to use channel 11 / PANID x0110 for all devices*/
45 //-DZDAPP_CONFIG_PAN_ID=0xFFFF
46 -DZDAPP_CONFIG_PAN_ID=0x0110
47
48 /* Minimum number of milliseconds to hold off the start of the device
49  * in the network and the minimum delay between joining cycles.
50  */
51 -DNWK_START_DELAY=100
52
53 /* Mask for the random joining delay. This value is masked with
54  * the return from osal_rand() to get a random delay time for
55  * each joining cycle.  This random value is added to NWK_START_DELAY.
56  * For example, a value of 0x007F will be a joining delay of 0 to 127
57  * milliseconds.
58  */
59 -DEXTENDED_JOINING_RANDOM_MASK=0x007F
60
61 /* Minimum number of milliseconds to delay between each beacon request
62  * in a joining cycle.
63  */
64 -DBEACON_REQUEST_DELAY=100
65
66 /* Mask for the random beacon request delay. This value is masked with the
67  * return from osal_rand() to get a random delay time for each joining cycle.
68  * This random value is added to DBEACON_REQUEST_DELAY. For example, a value
69  * of 0x00FF will be a beacon request delay of 0 to 255 milliseconds.
70  */
71 -DBEACON_REQ_DELAY_MASK=0x00FF
72
73 /* Jitter mask for the link status report timer. This value is masked with the
74  * return from osal_rand() to add a random delay to _NIB.nwkLinkStatusPeriod.
75  * For example, a value of 0x007F allows a jitter between 0-127 milliseconds.
76  */
77 -DLINK_STATUS_JITTER_MASK=0x007F
78
79 /* in seconds; set to 0 to turn off route expiry */
80 -DROUTE_EXPIRY_TIME=30
81
82 /* This number is used by polled devices, since the spec'd formula
83  * doesn't work for sleeping end devices.  For non-polled devices,
84  * a formula is used. Value is in 2 milliseconds periods
85  */
```

```
86 -DAPSC_ACK_WAIT_DURATION_POLLED=3000
87
88 /*  Default indirect message holding timeout value:
89  *  1-65535 (0 -> 65536) X CNT_RTG_TIMER X RTG_TIMER_INTERVAL
90  */
91 -DNWK_INDIRECT_MSG_TIMEOUT=7
92
93 /* The number of simultaneous route discoveries in network */
94 -DMAX_RREQ_ENTRIES=8
95
96 /* The maximum number of retries allowed after a transmission failure */
97 -DAPSC_MAX_FRAME_RETRIES=3
98
99 /* Max number of times retry looking for the next hop address of a message */
100 -DNWK_MAX_DATA_RETRIES=2
101
102 /* Number of times retry to poll parent before indicating loss of synchronization
103  * with parent. Note that larger value will cause longer delay for the child to
104  * rejoin the network.
105  */
106 -DMAX_POLL_FAILURE_RETRIES=2
107
108 /* The number of items in the broadcast table */
109 -DMAX_BCAST=9
110
111 /* The maximum number of groups in the groups table */
112 -DAPS_MAX_GROUPS=16
113
114 /* Number of entries in the regular routing table plus additional
115  * entries for route repair
116  */
117 -DMAX_RTG_ENTRIES=40
118
119 /* Maximum number of entries in the Binding table. */
120 -DNWK_MAX_BINDING_ENTRIES=4
121
122 /* Maximum number of cluster IDs for each binding table entry.
123  * Note that any value other than the default value may cause a
124  * compilation warning but Device Binding will function correctly.
125  */
126 -DMAX_BINDING_CLUSTER_IDS=4
127
128 /* Default security key. */
129 -DDEFAULT_KEY="{0x01, 0x03, 0x05, 0x07, 0x09, 0x0B, 0x0D, 0x0F, 0x00, 0x02, 0x04,
        0x06, 0x08, 0x0A, 0x0C, 0x0D}"
130
131 /* Reset when ASSERT occurs, otherwise flash LEDs */
132 //-DASSERT_RESET
133
134 /* Set the MAC MAX Frame Size (802.15.4 default is 102) */
135 -DMAC_MAX_FRAME_SIZE=116
136
137 /* Minimum transmissions attempted for Channel Interference detection,
138  * Frequency Agility can be disabled by setting this parameter to zero.
139  */
140 -DZDNWKMGR_MIN_TRANSMISSIONS=20
141
142 /* Compiler keywords */
143 -DCONST="const __code"
```

```
144 -DGENERIC=__generic
145
146 /*************************************
147  * The following are for End Devices only
148  *************************************/
149
150 -DRFD_RCVC_ALWAYS_ON=FALSE
151
152 /* The number of milliseconds to wait between data request polls to the
        coordinator. */
153 -DPOLL_RATE=1000
154
155 /* This is used after receiving a data indication to poll immediately
156  * for queued messages...in milliseconds.
157  */
158 -DQUEUED_POLL_RATE=100
159
160 /* This is used after receiving a data confirmation to poll immediately
161  * for response messages...in milliseconds
162  */
163 -DRESPONSE_POLL_RATE=100
164
165 /* This is used as an alternate response poll rate only for rejoin request.
166  * This rate is determined by the response time of the parent that the device
167  * is trying to join.
168  */
169 -DREJOIN_POLL_RAT
```

Apéndice B

Software de control propuesto para enviar y recibir mensajes y ejercer tareas de control

B.1. Listados para el ejecutable *escribiryenviarconhilos*

Listing B.1: Código escribiryenviarconhilos.c

```
1
2         //      .h para la programaci n con hilos,
3         //      que incluye todas las referencias necesarias
4 #include "writeinserialhilos.h"
5
6
7
8 /*       this version uses libraries
9  *       compilation     gcc -o escribiryenviarconhilos
10 *       escribiryenviarconhilos.c funcioneswriteinserialhilos.c
11 *       -lzigbeeti -pthread     */
12 *       ........................*/
13 *       -lzigbee is because it uses the libzigbee.so.1.0
14 *              lib in /usr/lib                          */
15
16
17 int main(int argc, char *argv[])
18 {
19        /*      Variables               */
20        pthread_t h1,h2,h3;
21        pthread_attr_t  a1;
22
23        /*      inicializaciones        */
24        pthread_attr_init(&a1);
25
26
27
28        nmensajesrecibidos = 0;
29        nmensajesaenviar = 0;
30        nmensajesefectivamenteenviados = 0;
31        nmensajesefectivamenteprocesados = 0;
```

```
32
33          /*      Ejecucion       */
34          //      primero se introduce el dispositivo en el que enviar
35
36          printf("Dispositivo a conectar ,por ejemplo /dev/ttyUSB0):");
37          scanf("%s" , entrada_dispositivo);
38
39          pthread_create(&h1 , &a1 , thread_recibemensajes , NULL );
40          pthread_create(&h2 , &a1 , thread_codigoprincipal , NULL );
41          pthread_create(&h3 , &a1 , thread_enviamensajes , NULL );
42
43          //      esperamos que los hilos lleguen al final antes de acabar
44          //      aunque en realidad hay dos que implican un bucle infinito
45
46          pthread_join(h1, NULL);
47          pthread_join(h2, NULL);
48          pthread_join(h3, NULL);
49
50          return 0;
51  }
```

Listing B.2: Código funcioneswriteinserialhilos.c

```
1  #include "writeinserialhilos.h"
2
3
4  void *thread_codigoprincipal(void* p)
5  {
6
7                  //      Variables locales
8                  char localentrada_datos[LONGITUDDATOS + 1];             // el +1
                       es para que haga sitio para '\0'
9                  int localnmensajesrecibidos = 0;
10                 int localnmensajesprocesados;                           // puede
                       que esta variable no se use
11                 unsigned int localcomandorecibido;
12                 //char localdatosasciirecibidos[LONGITUDDATOS + 1];     // el +1
                       es para que haga sitio para '\0'
13                 int localnumerodemensajesencolados = 0;                 // el
                       numero de mensajes que se llevan encolados
14                 char* plocaldatosasciirecibidos;                        // en vez
                       de char localdatosasciirecibidos[]
15                 int i  = 0;
16
17                 //      0
18                 //      mientras no se reciban mensajes entrantes, no se puede
                       continuar
19                 //      porque no se sabe a d nde enviar
20                 while(localnmensajesrecibidos == 0)
21                 {
22                         /*      INICIO SECCION CRITICA  */
23                         pthread_mutex_lock(&m1);
24
25                         localnmensajesrecibidos = nmensajesrecibidos;
26
27                         pthread_mutex_unlock(&m1);
28                         /*      FIN SECCION CRITICA     */
29                                 sleep(SEGUNDOSSLEEP);
30
```

```
31                  printf("thread_codigoprincipal: esperando a que el robot
                        comunique que est  listo\n");
32              }
33
34              /*      INICIO SECCION CRITICA  */
35              pthread_mutex_lock(&m1);
36
37              if(localnmensajesrecibidos > 0)
38              {
39                      /*      hay que rellenar
40
41                              unsigned int entrada_direcciondestino;
42                              unsigned int entrada_endpointdestino;
43
44                              con los datos del primer mensaje recibido
45                              y luego enviar un mensaje, incrementando
46                              nmensajesefectivamenteenviados
47                      */
48                      entrada_direcciondestino = (unsigned int)(
                            mensajesrecibidos[0][1] & 0x00FF) | ((mensajesrecibidos
                            [0][2] << 8) & 0xFF00 );
49                      entrada_endpointdestino = (unsigned int)(mensajesrecibidos
                            [0][6]);
50
51                      localnmensajesprocesados++;             //      llevamos 1
                             mensaje entrante procesado
52                      //      nmensajesaenviar = localnumerodemensajesencolados;
53                      nmensajesefectivamenteprocesados =
                            localnmensajesprocesados;
54
55
56                      printf("thread_codigoprincipal: el robot ya ha comunicado
                            que est  listo\n");
57                      printf("thread_codigoprincipal: direccion a responder %2.2
                            x%2.2x \n",mensajesrecibidos[0][2],mensajesrecibidos
                            [0][1]);
58                      printf("thread_codigoprincipal: endpoint a responder %2.2x
                             \n",mensajesrecibidos[0][6]);
59              }
60              pthread_mutex_unlock(&m1);
61              /*      FIN SECCION CRITICA     */
62              //else{;}       //      hay mensajes para enviar y ya se tiene el
                    destino y ep
63
64
65
66
67
68              //      I
69              /*      ENVIA mensaje 1 con Tarea 1: TRASLACION                 */
70              //      Primero se escribe en el buffer mensajesaenviar[][]
71
72              /*      INICIO SECCION CRITICA  */
73              pthread_mutex_lock(&m1);
74
75
76
77              localentrada_datos[0] =         '0';
78              localentrada_datos[1] =         '1';
```

```
79                  localentrada_datos[2] =         '5';
80                  localentrada_datos[3] =         '0';
81                  localentrada_datos[4] =         '0';
82                  localentrada_datos[5] =         '0';
83                  localentrada_datos[6] =         ':';
84                  localentrada_datos[7] =         '0';
85                  localentrada_datos[8] =         '0';
86                  localentrada_datos[9] =         '0';
87                  localentrada_datos[10] =        '0';
88                  localentrada_datos[11] =        '0';
89                  localentrada_datos[12] =        '0';
90                  localentrada_datos[13] =        ':';
91                  localentrada_datos[14] =        '0';
92                  localentrada_datos[15] =        '0';
93                  localentrada_datos[16] =        '0';
94                  localentrada_datos[17] =        '0';
95                  localentrada_datos[18] =        '0';
96                  localentrada_datos[19] =        '0';
97                  localentrada_datos[20] =        '\0';
98
99
100
101
102
103                 zigbeeti_rellenaBufferConInfo(entrada_direcciondestino ,
                        entrada_endpointdestino ,
104                 (unsigned int) 0x0A , (unsigned int) COM_TRASLACIONABSOLUTA ,
                        localentrada_datos ,
105                 &mensajesaenviar[0][0]  );
106
107                 localnumerodemensajesencolados++;
108                 nmensajesaenviar = localnumerodemensajesencolados;
109                 //      llevamos 1 mensaje encolado (asumimos que solo este
                        proceso encola)
110
111                 pthread_mutex_unlock(&m1);
112                 /*      FIN SECCION CRITICA     */
113
114
115                 printf("thread_codigoprincipal ha encolado el mensaje n mero %d
116                         con comando %2.2x y con datos %s \n",
                                localnumerodemensajesencolados,
117                         COM_TRASLACIONABSOLUTA, &mensajesaenviar[0][10]);
118
119                 printf("thread_codigoprincipal nmensajesaenviar :%d \n",
                        nmensajesaenviar);
120                 printf("thread_codigoprincipal nmensajesrecibidos :%d \n",
                        nmensajesrecibidos);
121                 printf("thread codigoprincipal nmensajesefectivamenteenviados :%d
                        \n",
122                         nmensajesefectivamenteenviados);
123                 printf("thread_codigoprincipal nmensajesefectivamenteprocesados :%
                        d \n",
124                         nmensajesefectivamenteprocesados);
125
126                 /*      Espera RECEPCION confirmaci n finalizacion Tarea 1      */
127
128                 // Mientras no haya ning n mensaje recibido esperar
129                 while(localnmensajesrecibidos <= 1)
```

```
130                 {
131                         /*      INICIO SECCION CRITICA  */
132                         pthread_mutex_lock(&m1);
133
134                         localnmensajesrecibidos = nmensajesrecibidos;
135
136                         pthread_mutex_unlock(&m1);
137                         /*      FIN SECCION CRITICA     */
138                         sleep(SEGUNDOSSLEEP);
139                 }
140
141                 // Procesamos el mensaje
142
143                 localnmensajesprocesados++;
144                 //      llevamos 2 mensajes entrantes procesados
145
146                 /*      INICIO SECCION CRITICA  */
147                 pthread_mutex_lock(&m1);
148
149                 //      Aqu  habria que seguir una rutina para extraer los datos
                        del mensa
150                 nmensajesefectivamenteprocesados = localnmensajesprocesados;
151                 localcomandorecibido = (unsigned int)(mensajesrecibidos[
                        localnmensajesprocesados -1][4] & 0x00FF)
152                 | ((mensajesrecibidos[localnmensajesprocesados -1][5] << 8) & 0
                        xFF00 );
153
154                 plocaldatosasciirecibidos = &mensajesrecibidos[
                        localnmensajesprocesados -1][10];
155
156                 //      Procesar los datos (en esta rutina NO se procesan los
                        datos) -- podriamos sacarlos por pantalla,pero
157                 //      fuera de la secci n cr tica
158                 // nmensajesefectivamenteprocesados++;
159
160                 pthread_mutex_unlock(&m1);
161                 /*      FIN SECCION CRITICA     */
162
163
164                 //      II
165                 ...
166                 //      III
167                 ...
168                 //      IV
169                 ...
170                 //      V
171                 ...
172                 //      VI
173
174 }
175
176
177 //      FUNCION/Thread
178 //      esta funci n ir  enviando los mensajes que el thread
179 //      thread_codigoprincipal coloca en el buffer de salida,
180 //      a la direccion/EP/cluster de origen del primer mensaje del
181 //      buffer de mensajes de entrada
182
183 void *thread_enviamensajes(void* p)
```

```
184 {
185         int i;
186         unsigned int direcciondestino;
187         unsigned int endpointdestino;
188         unsigned int clusterdestino;
189         unsigned int comando;
190         char cadenaasciidatos[LONGITUDDATOS + 1];
191         char* puertoserie;
192
193
194         while(1)
195         {
196
197                 /*      Una vez ya se ha recibido un mensaje entrante el proceso
                            ya determina
198                         la direcci n de la respuesta y la integra en el mensaje
                                de respuesta.
199                         A partir de este punto obtenemos los datos del buffer para
                                 enviar
200                         mensajes mientras nmensajesaenviar <
                                nmensajesefectivamenteenviados
201                         //  0       0xFF
202                         //  1-2     origin node address
203                         //  3       EP origin
204                         //  4-5     destination node address
205                         //  6       EP destination
206                         //  7       Destination cluster
207                         //  8-9 16-bit command
208                         //  10-30 21 bytes of data ended in '\0' (en realidad 20)
209                         //  31      '\0'
210                 */
211
212
213                 /*      INICIO SECCION CRITICA  */
214                 pthread_mutex_lock(&m1);
215
216
217
218
219                 if(nmensajesaenviar > nmensajesefectivamenteenviados)
220                 {
221
222                         printf("thread_enviarmensajes PRINCIPIO del bucle if \n");
223
224                         /*      extraer los campos      */
225
226
227                         direcciondestino =
228                                 (unsigned int)(mensajesaenviar[
                                        nmensajesefectivamenteenviados][4] & 0x00FF) |
229                                 ((mensajesaenviar[nmensajesefectivamenteenviados
                                        ][5] << 8) & 0xFF00 );
230                         endpointdestino =
231                                 (unsigned int)(mensajesaenviar[
                                        nmensajesefectivamenteenviados][6]);
232                         clusterdestino =
233                                 (unsigned int)(mensajesaenviar[
                                        nmensajesefectivamenteenviados][7]);
234                         comando =
```

```
235                                     (unsigned int)(mensajesaenviar[
                                            nmensajesefectivamenteenviados][8] & 0x00FF) |
236                                     ((mensajesaenviar[nmensajesefectivamenteenviados
                                            ][9] << 8) & 0xFF00 );
237
238
239
240                             for(i = 10;i < (10 + LONGITUDDATOS + 1); i++)
241                             {
242                                     *(cadenaasciidatos + i - 10) =
243                                     *(mensajesaenviar + nmensajesefectivamenteenviados
                                             * LONGITUDBUFFERSERIE + i);
244                             }
245
246
247                             /*      usar funcion enviar     */
248
249
250                             zigbeeti_enviar(direcciondestino , endpointdestino ,
251                                     clusterdestino , comando , cadenaasciidatos,
                                            entrada_dispositivo);
252                             nmensajesefectivamenteenviados++;
253
254                             printf("thread_enviarmensajes FINAL del bucle if \n");
255
256                             printf("thread_enviarmensajes nmensajesaenviar :%d \n",
                                    nmensajesaenviar);
257                             printf("thread_enviarmensajes nmensajesrecibidos :%d \n",
                                    nmensajesrecibidos);
258                             printf("thread enviarmensajes
                                    nmensajesefectivamenteenviados :%d \n",
259                                         nmensajesefectivamenteenviados);
260                             printf("thread_enviarmensajes
                                    nmensajesefectivamenteprocesados :%d \n",
261                                         nmensajesefectivamenteprocesados);
262                     }
263
264                     pthread_mutex_unlock(&m1);
265                     /*      FIN SECCION CRITICA     */
266             }
267             pthread_exit(0);
268     }
269
270
271
272
273
274
275     void *thread_recibemensajes(void* p)
276     {
277
278
279             //      Local variables
280             unsigned char pmensajearecibir[LONGITUDBUFFERSERIE];
281             int i;
282
283             while(1)
284             {
285                     zigbeeti_recibir(entrada_dispositivo , pmensajearecibir);
```

```
286
287
288                 /*      PRINCIPIO SECCION CRITICA PROTEGIDA POR MUTEX m1 */
289                 pthread_mutex_lock(&m1);
290
291                 // 1)   Se escribe el mensaje en la variable y se incrementa el
                        contador
292
293                 for(i = 0;i < LONGITUDBUFFERSERIE; i++)
294                 {
295                         *(&mensajesrecibidos[0][0] + nmensajesrecibidos *
                            LONGITUDBUFFERSERIE + i) =
296                                 *(&pmensajearecibir[0] + i);
297                 }
298
299                 nmensajesrecibidos++;
300
301                 pthread_mutex_unlock(&m1);
302                 /*      FIN SECCION CRITICA PROTEGIDA POR MUTEX m1 */
303
304
305
306
307                 //      escribir lo que se ha recibido
308                 for(i=0;i < LONGITUDBUFFERSERIE; i++)
309                 {
310                         printf("%2.2x",*(pmensajearecibir + i));
311                 }
312                 printf("\n");
313
314                 printf("thread_recibemensajes nmensajesaenviar :%d \n",
                        nmensajesaenviar);
315                 printf("thread_recibemensajes nmensajesrecibidos :%d \n",
                        nmensajesrecibidos);
316                 printf("thread_recibemensajes nmensajesefectivamenteenviados :%d \
                        n",
317                         nmensajesefectivamenteenviados);
318                 printf("thread_recibemensajes nmensajesefectivamenteprocesados :%d
                         \n",
319                         nmensajesefectivamenteprocesados);
320
321         }
322         pthread_exit(0);
323
324 }
```

Listing B.3: Código writeinserialhilos.c

```
1
2 #ifndef WRITEINSERIALCONHILOS_H
3 #define WRITEINSERIALCONHILOS_H
4
5 #ifdef __cplusplus
6 extern "C" {
7 #endif
8
9
10
11
```

```
12 // header para libreria zigbeeti
13 #include "writeinserial.h"
14
15 //      UNIX and Standard C includes
16 #include <stdio.h>   /* Standard input/output definitions */
17 #include <string.h>  /* String function definitions */
18 #include <unistd.h>  /* UNIX standard function definitions */
19 #include <fcntl.h>   /* File control definitions */
20 #include <errno.h>   /* Error number definitions */
21 #include <termios.h> /* POSIX terminal control definitions */
22 #include <sys/syscall.h>
23 #include <sys/types.h>
24 #include <pthread.h>    /*      Para usar threads       */
25
26 /****           defs                       ****/
27 #define NUMEROMENSAJESBUFFERRECEPCION   1000
28 #define NUMEROMENSAJESBUFFERENVIO       1000
29 #define SEGUNDOSSLEEP                   1
30
31 #define MAXIMONUMEROMENSAJESAENVIAR     10000
32 #define MAXIMONUMEROMENSAJESARECIBIR    10000
33
34
35 //      COMANDOS (son unsigned int de 2 bytes, expresados en hexadecimal )
36
37 #define COM_PRESENCIAGENERICA                       0x0000  // Se ha cambiado el
       c digo del CC2530
38 #define COM_CONFIRMACIONRECEPCION                   0x0001
39 #define COM_CONFIRMACIONEJECUCIONCORRECTA           0x0002
40 #define COM_CONFIRMACIONEJECUCIONINCORRECTA         0x0003
41 #define COM_TRASLACIONABSOLUTA                      0x0004
42 #define COM_TRASLACIONRELATIVA                      0x0005
43 #define COM_TRASLACIONROTACIONABSOLUTA              0x0006
44 #define COM_TRASLACONROTACIONRELATIVA               0x0007
45 #define COM_ROTACIONABSOLUTA                        0x0008
46 #define COM_ROTACIONRELATIVA                        0x0009
47 #define COM_POSICIONAMIENTOPINZA                    0x000A
48
49 #define DEV_ORDENCOMPLETADA                         0x000B
50
51
52
53 /****           global variables        ****/
54 //      hilos
55 //      las variables se definen en el main
56 //      mutex
57 pthread_mutex_t m1;
58 //      pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
59
60 //      LAS SIGUIENTES VARIABLES SIRVEN PARA LA COMUNICACION
61 //      ENTRE
62 //      buffers envio y recepcion
63 unsigned char mensajesrecibidos[NUMEROMENSAJESBUFFERRECEPCION][LONGITUDBUFFERSERIE
       ];
64 unsigned char mensajesaenviar[NUMEROMENSAJESBUFFERENVIO][LONGITUDBUFFERSERIE];
65
66 //      Para llevar la cuenta, porque C no tiene containers
67 int nmensajesrecibidos;
68 int nmensajesaenviar;
```

```
69 int nmensajesefectivamenteenviados ;            // Hay una diferencia entre los
       mensajes enviados y los que hay en el buffer a enviar
70 int nmensajesefectivamenteprocesados;   // Hay una diferencia entre los mensajes
       recibidos y los que el thread_codigoprincipal ha procesado
71
72
73 //      Otras variables para comunicarse entre hilos
74
75 char entrada_dispositivo[MAXIMALINEA]; //      esperamos algo como "ttyUSB0"
76 unsigned int entrada_direcciondestino;          //      el dispositivo en espera
       recibir  un mensaje peri dico del cual sacar  el origen, ep y lo pasar  al
       hilo adecuado
77 unsigned int entrada_endpointdestino;   //      esperamos un byte
78
79
80
81
82
83 /****   prototipos funciones hilos      ****/
84 void *thread_codigoprincipal(void* p);
85 void *thread_enviamensajes(void* p);
86 void *thread_recibemensajes(void* p);
87
88
89
90
91
92 #ifdef __cplusplus
93         }
94 #endif
95
96
97 #endif // end WRITEINSERIALCONHILOS_H
```

B.2. Listados para la librería *zigbeeti*

Listing B.4: Código escribiryenviar.c

```
1 //      Project specific includes
2 #include "writeinserial.h"
3
4 //      UNIX and Standard C includes
5 #include <stdio.h>   /* Standard input/output definitions */
6 #include <string.h>  /* String function definitions */
7 #include <unistd.h>  /* UNIX standard function definitions */
8 #include <fcntl.h>   /* File control definitions */
9 #include <errno.h>   /* Error number definitions */
10 #include <termios.h> /* POSIX terminal control definitions */
11 #include <sys/syscall.h>
12 #include <sys/types.h>
13
14
15 /*      generate the dynamic libraries (so objects)
16
17         gcc -Wall -fPIC -c functionswriteinserial.c escribiryenviar.c
```

```
18          gcc -shared -Wl, -soname, libzigbeeti.so.1 -o libzigbeeti.so.1.0
                functionswriteinserial.o escribiryenviar.o
19
20          en el directorio donde se ponga, hay que crear DOS symbolic links
21
22          ln -sf /path-to-library-directory/libzigbeeti.so.1.0 /path-to-link-
                directory/libzigbeeti.so
23          ln -sf /path-to-library-directory/libzigbeeti.so.1.0 /path-to-link-
                directory/libzigbeeti.so.1
24
25  */
26
27
28
29  //      FUNCION
30  //      cada vez que se invoca devuelve UNA trama de LONGITUDBUFFERSERIE octetos
31  //      as  que cada vez que para recibir constantemente hay que meter esta
        funci n
32  //      en alguna estructura iterativa
33  //      puertoserie             cadena del tipo /dev/ttyUSB0"
34  //      pmensajearecibir        puntero de tipo unsigned char* con
        LONGITUDBUFFERSERIE bytes reservados
35  //                              en la llamada.
36  void zigbeeti_recibir(char* puertoserie, unsigned char* pmensajearecibir)
37  {
38
39          //      Variables locales
40          int fd;         //      Port handler
41          int nbytes;     //      no bytes enviados
42
43          //      se abre el puerto
44          fd = zigbeeti_open_port_bloqueante(puertoserie);
45          read(fd, pmensajearecibir , 1);                         // POSIX
46
47          //      comprueba que el primer byte de la trama es 0xFF
48          while(*pmensajearecibir != 0xFF){read(fd, pmensajearecibir , 1);}
49          nbytes = 1;
50
51          for(nbytes = 1; nbytes < LONGITUDBUFFERSERIE; nbytes++)
52          {
53                  read(fd , pmensajearecibir + nbytes, 1 );       // POSIX
54          }
55  }
56
57
58
59
60
61
62  //      FUNCION
63  //      se usa para generar una trama de 32 octetos a enviar, suministrando la
        siguiente informaci n
64  //      unsigned int direcciondestino   --      direcci n de destino que se
        convertir  en 2 bytes
65  //      unsigned int endpointdestino    --      endpoint (1 byte) de destino
66  //      unsigned int clusterdestino     --      cluster de destino, en caso de
        usarse (1 byte)
67  //      unsigned int comando            --      comando (2 bytes)
```

```
68 //      char* cadenaasciidatos          --      puntero a un string que encapsula
       los datos a enviar
69 //      car* puertoserie                --      puntero a un string que describa
       el puerto a abrir
70 void zigbeeti_enviar(unsigned int direcciondestino, unsigned int endpointdestino,
       unsigned int clusterdestino, unsigned int comando, char* cadenaasciidatos,char*
        puertoserie)
71 {
72
73         //      local variables
74         //int i;
75         int fd;                                                         //
               Port handler
76         int nbytes;                                                     //      no
                bytes enviados
77         //unsigned char mensajeaenviar[LONGITUDBUFFERSERIE];
78         //unsigned char pmensajeaenviar[LONGITUDBUFFERSERIE];           //
               inicializado como array para reservar LONGITUDBUFFERSERIE
79         unsigned char pmensajeaenviar[LONGITUDBUFFERSERIE];
                                                                //      bytes
80
81         //      rellena el buffer apuntado por pmensajea enviar
82
83         zigbeeti_rellenaBufferConInfo(direcciondestino , endpointdestino ,
               clusterdestino , comando ,   cadenaasciidatos, pmensajeaenviar);
84
85         //for(i = 0; i < LONGITUDBUFFERSERIE; i++)
86         //{
87         //      mensajeaenviar[i] = *(pmensajeaenviar + i);
88         //}
89         //      verifica el tratamiento de los datos
90         zigbeeti_verificaloqueseenvia(pmensajeaenviar);
91         //      se abre el puerto
92         fd = zigbeeti_open_port_bloqueante(puertoserie);
93         //      se escribe en el puerto
94         // nbytes = write(fd, &mensajeaenviar[0], LONGITUDBUFFERSERIE); // POSIX
95         nbytes = write(fd , pmensajeaenviar , LONGITUDBUFFERSERIE);
96         //      se cierra el puerto
97         close(fd);                                                      // POSIX
98 }
```

Listing B.5: Código functionswriteinserial.c

```
1  //      Project specific includes
2  #include "writeinserial.h"
3
4  //      UNIX and Standard C includes
5  #include <stdio.h>   /* Standard input/output definitions */
6  #include <string.h>  /* String function definitions */
7  #include <unistd.h>  /* UNIX standard function definitions */
8  #include <fcntl.h>   /* File control definitions */
9  #include <errno.h>   /* Error number definitions */
10 #include <termios.h> /* POSIX terminal control definitions */
11 #include <sys/syscall.h>
12 #include <sys/types.h>
13
14
15
16 //      FUNCION
```

```
17 //      opens the PUERTOSERIEOBJETIVO defined as a chain in writeinserial.h
18 //      blocking function
19 int zigbeeti_open_port_bloqueante(char* puertoserie)
20 {
21         int fd;
22         struct termios options;
23
24
25         //      El puerto serie se especifica al arrancar el di logo
26         fd = open(puertoserie, O_RDWR );
27
28         if(fd == -1)
29         {
30                 //      Could not open the port
31                         perror("open_port: unable to open the port ");
32
33         }
34         else
35         {
36                 tcgetattr(fd, &options);
37
38                 cfsetispeed(&options, B9600);
39                 cfsetospeed(&options, B9600);
40
41                 options.c_cflag &= ~PARENB;
42                 options.c_cflag &= ~CSTOPB;
43                 options.c_cflag &= ~CSIZE;
44                 options.c_cflag |= ~CS8;
45
46                 options.c_iflag |= (IXON|IXOFF|IXANY);
47
48                 tcflush(fd, TCIFLUSH);
49                 tcflush(fd, TCOFLUSH);
50
51                 fcntl(fd, F_SETFL, 0);
52
53                 tcsetattr(fd, TCSANOW, &options);
54
55                 return(fd);
56         }
57 }
58 // Nota: la siguiente version se podria usar si el puerto
59 // se preconfigurara con una terminal, como putty
60 /*
61 int zigbeeti_open_port_bloqueante(char* puertoserie)
62 {
63         int fd;
64
65         //      El puerto serie se especifica al arrancar el di logo
66         fd = open(puertoserie, O_RDWR );        // POSIX
67
68         if(fd == -1)
69         {
70                 //      Could not open the port
71                 perror("open_port: unable to open the port ");
72         }
73         else{fcntl(fd, F_SETFL, 0);}
74
75         return(fd);
```

```
76 }
77 */
78
79 //      FUNCION
80 //      Rellena el buffer variable global a enviar serie CON LOS DATOS
       SUMINISTRADOS
81 //      pmensajeaenviar         puntero unsigned char* con LONGITUDBUFFERSERIE
       bytes libres
82 void zigbeeti_rellenaBufferConInfo(unsigned int entrada_direcciondestino ,
       unsigned int entrada_endpointdestino , unsigned int entrada_clusterdestino ,
       unsigned int entrada_comando ,  char* entrada_datos, unsigned char*
       pmensajeaenviar)
83 {
84
85
86         //      Local variable
87         int i;
88         char direcciondestino_LSB;
89         char direcciondestino_MSB;
90         char endpointdestino;
91         char clusterdestino;
92         char comando_LSB;
93         char comando_MSB;
94
95
96         direcciondestino_LSB = (char) ( entrada_direcciondestino & 0x000000FF );
97         direcciondestino_MSB = (char) ((entrada_direcciondestino >> 8) & 0
               x000000FF );
98         endpointdestino = (char) ( entrada_endpointdestino & 0x000000FF );
99         clusterdestino = (char) ( entrada_clusterdestino & 0x000000FF );
100        comando_LSB = (char) ( entrada_comando & 0x000000FF);
101        comando_MSB = (char) ((entrada_comando >> 8) & 0x000000FF);
102
103
104
105        *pmensajeaenviar  = (char) 0xFF;        // el primero SIEMPRE ES 0xFF
106        *(pmensajeaenviar + 1)  = (char) 0x00;  // dir. ORIGEN LSB - lo rellena el
               SoC
107        *(pmensajeaenviar + 2)  = (char) 0x00;  // dir. ORIGEN MSB - lo rellena el
               SoC
108        *(pmensajeaenviar + 3)  = (char) 0x00;  // EP.  ORIGEN - lo rellena el SoC
109        *(pmensajeaenviar + 4)  = direcciondestino_LSB; // dir. DESTINO LSB - SE
               RELLENA AQU
110        *(pmensajeaenviar + 5)  = direcciondestino_MSB; // dir. DESTINO MSB - SE
               RELLENA AQU
111        *(pmensajeaenviar + 6)  = endpointdestino;              // EP.  DESTINO -
               SE RELLENA AQU
112        *(pmensajeaenviar + 7)  = clusterdestino;               // Cluster.
               DESTINO - no se usa
113        *(pmensajeaenviar + 8)  = (char) comando_LSB;   // Comando LSB
114        *(pmensajeaenviar + 9)  = (char) comando_MSB;   // Comando MSB
115
116        for(i = 10;i < (10 + LONGITUDDATOS + 1); i++)
117        {
118                *(pmensajeaenviar + i) = *(entrada_datos + i - 10);
119        }
120
121        *(pmensajeaenviar + 31) = '\0';
122
```

```
123         //      pmensajeaenviar es un puntero conocido, desde la funci n
                invocante, por lo que no
124         //      es necesario devolver nada
125
126 }
127
128
129
130 //      FUNCION
131 //      Imprime los datos interpretados ANTES de abrir el puerto
132 void zigbeeti_verificaloqueseenvia(char* mensajeaenviar)
133 {
134         int i;
135         for(i = 0; i < LONGITUDBUFFERSERIE;i++)
136         {
137                 printf("byte en posicion %2.2i, valor %2.2x expresado en
                        hexadecimal\n", i ,*(mensajeaenviar + i));
138         }
139
140 }
```

Listing B.6: Código writeinserial.h

```
1  #ifndef WRITEINSERIAL_H
2  #define WRITEINSERIAL_H
3
4  #ifdef __cplusplus
5  extern "C" {
6  #endif
7
8
9  /*      Includes        */
10 #include <sys/syscall.h>
11 #include <sys/types.h>
12
13
14 /*      definitions     */
15 #define PUERTOSERIEOBJETIVO                    "/dev/ttyUSB0"
16 #define LONGITUDBUFFERSERIE                    32
17 #define MAXIMALINEA                            100
18 #define MAXIMALONGITUDDIRECCIONES              2
19 #define MAXIMALONGITUDNOMBREPUERTOSERIE        50
20 #define LONGITUDDATOS                          20
21
22
23 /*      Global variables        */
24
25
26 /*      prototipos funciones    */
27 int zigbeeti_open_port_bloqueante(char* puertoserie);
28
29 void zigbeeti_rellenaBufferConInfo(unsigned int entrada_direcciondestino ,
       unsigned int entrada_endpointdestino , unsigned int entrada_clusterdestino ,
       unsigned int entrada_comando ,  char* entrada_datos, unsigned char*
       pmensajeaenviar);
30
31 void zigbeeti_verificaloqueseenvia(char* mensajeaenviar);
32
33
```

```
34 //      FUNCION
35 //      se usa para generar una trama de 32 octetos a enviar, suministrando la
       siguiente informaci n
36 //      unsigned int direcciondestino   --      direcci n de destino que se
       convertir  en 2 bytes
37 //      unsigned int endpointdestino    --      endpoint (1 byte) de destino
38 //      unsigned int clusterdestino     --      cluster de destino, en caso de
       usarse (1 byte)
39 //      unsigned int comando            --      comando (2 bytes)
40 //      char* cadenaasciidatos          --      puntero a un string que encapsula
       los datos a enviar
41 //      car* puertoserie                --      puntero a un string que describa
       el puerto a abrir
42 void zigbeeti_enviar(unsigned int direcciondestino, unsigned int endpointdestino,
       unsigned int clusterdestino, unsigned int comando, char* cadenaasciidatos,char*
        puertoserie);
43
44
45 //      FUNCION
46 //      cada vez que se invoca devuelve UNA trama de LONGITUDBUFFERSERIE octetos
47 //      as  que cada vez que para recibir constantemente hay que meter esta
       funci n
48 //      en alguna estructura iterativa
49 //      puertoserie             cadena del tipo /dev/ttyUSB0"
50 //      pmensajearecibir        puntero de tipo unsigned char* con
       LONGITUDBUFFERSERIE bytes reservados
51 //                              en la llamada.
52 void zigbeeti_recibir(char* puertoserie, unsigned char* pmensajearecibir);
53
54 /*      Types   */
55
56
57 #ifdef __cplusplus
58         }
59 #endif
60
61
62 #endif // end WRITEINSERIAL_H
```

Bibliografía

[1] dsPIC30F programmer's reference manual. on-line pdf, 2003.

[2] dsPIC30F Family Reference Manual. on-line pdf, 2005.

[3] Z-stack OS Abstraction Layer API. on-line pdf, 2006.

[4] 802.15.4 MAC Application Programming Interface. on-line pdf, 2009.

[5] Application note: method for discovering network topology. on-line pdf, 2009.

[6] CC253x System-on-chip solution for 2.4-GHz IEEE 802.15.4 and ZigBee Applications: User Guide. on-line pdf, 2009.

[7] HAL drivers API. on-line pdf, 2009.

[8] HAL porting guide. on-line pdf, 2009.

[9] Z-stack compile options. on-line pdf, 2009.

[10] IAR Embedded Workbench: IDE Project Management and building Guide. on-line pdf, 2010.

[11] Microchip 16-Bit Language Tools libraries. on-line pdf, 2010.

[12] SmartRF05 Evaluation Board User's guide. on-line pdf, 2010.

[13] ERIKA Enterprise manual for the Microchip PIC24, dsPIC30 (R) DSC and dsPIC33 (R) DSC targets: the RTOS for PIC devices . on-line pdf, 2011.

[14] Z-stack developer's guide. on-line pdf, 2011.

[15] Peter Baer Galvin Greg Gagne Abraham Silberschatz. *Operating System Concepts*. John Wiley and Sons, Ltd, 2005.

[16] Jan Axelson. *Serial Port Complete: COM ports, virtual COM ports and ports for embedded systems*. Lakeview Research LLC, 2007.

[17] Michael Barr. *Programming Embedded systems in C and C++*. O'Reilly, 1999.

[18] Giovanni Caire Dominic Greenwood Fabio Bellifemine. *Developing multi-agents with JADE*. John Wiley and Sons, Ltd, 2007.

[19] Machtelt Garrels. *Introduction to Linux: a hands on guide*. GNU press, 2004.

[20] Phillip A. Laplante. *Real time systems design and analysis*. IEEE Press Wiley Interscience, 2004.

[21] Wolfrang Mauerer. *Professional Linux Kernel architecture*. Wiley Publishing, Inc, 2008.

[22] Brian W. Kernighan Dennis M.Ritchie. *The C programming Language*. Prentice Hall, 1988.

[23] Kay A. Robbins Steven Robbins. *Unix systems programming: communication, concurrency and threads*. Prentice Hall, 2003.

[24] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. GNU press, 2010.

[25] Dirk Pensky Ulrich Karras and Octavio Rojas. Mobile Robotics in Education and Research of Logistics.

[26] R.C. Weber and M. Bellenberg. Robotino Manual, 2010.