Redes de Sensores Inalámbricos Autoalimentados

por Leandro G. Vacirca

Director de Tesis: MSc.Ing. Ricardo Vecchio

Facultad de Ciencias Fisicomatemáticas e Ingeniería

Pontificia Universidad Católica Argentina

20 de diciembre de 2011

Buenos Aires, Argentina

Agradecimientos

A nuestro tutor de tesis, Ing. Ricardo Vecchio, que sin su continuo apoyo, énfasis y su guía no hubiese sido posible haber alcanzado los objetivos propuestos en tiempo y forma.

A nuestro coordinador de Trabajo Final, Ing. Norberto Heyaca, por habernos dado el soporte para iniciar el trabajo y ayudarnos a encararlo de la mejor forma posible.

A mi amigo y compañero de tesis, Damián Iglesias, con quién hemos atravesado toda la carrera y culminamos un largo recorrido, realizando el trabajo final juntos.

A mi familia, mi mamá Dorita, mi hermana Estefy, mi sobrina Martina, mis tíos Ricky y Zule, mi ahijada Julieta y su hermanita Berenice, mi madrina Miriam, mis primas y primos, y mis dos abuelos José's, por su apoyo incondicional y constante respaldo.

A mi padre, y mis abuelas, que desde algún lugar estarán mirando...

A Marce, mi novia, a quién he conocido en nuestra facultad, y con quién he compartido maravillosos momentos en la carrera pero fundamentalmente por tanta paciencia y apoyo a lo largo de todos estos años.

Por último, a mi otro yo por haber soportado llegar hasta aquí!

Objetivo

El objetivo de la tesis es realizar una demostración de una aplicación real de una red de sensores inalámbricos autoalimentados (del inglés, Wireless Sensors Network). Teniendo en cuenta que se trata de un tema de gran actualidad en el mundo académico e industrial, se quiere implementar una plataforma genérica de hardware y software para el desarrollo de aplicaciones escalables. Se deberán introducir todos los conceptos necesarios para demostrar la factibilidad técnica de una red de sensores inalámbricos autoalimentados para la medición de temperatura, mediante la utilización de técnicas de *energy harvesting*.

Se pretende demostrar en consecuencia, la posibilidad del desarrollo de aplicaciones de ultra bajo consumo energético, como eje de partida para la implementación de redes inalámbricas de sensores que sean sustentables mediante energías presentes en el ambiente.

Por último, se desea dejar una puerta abierta para futuras implementaciones e investigaciones tomando como punto de partida al presente trabajo.

Introducción

En los últimos años estamos asistiendo a un incremento vertiginoso de la presencia de las comunicaciones inalámbricas en nuestra sociedad. Así, existen diferentes tecnologías que utilizamos cada día dependiendo de la aplicación a la que estén destinadas. No sólo el teléfono móvil se ha convertido en imprescindible, también para muy corto alcance se está imponiendo el uso de Bluetooth, por ejemplo. Existen multitud de dispositivos como PDAs, manos libres, vehículos, etc. que ya disponen de esta tecnología.

Por otro lado, si queremos disponer de comunicaciones en un ámbito local sin necesidad de cables, existen diferentes tecnologías como las Wi-Fi. En definitiva, la sociedad va descubriendo las ventajas de los entornos inalámbricos y en un futuro próximo aparecerán nuevas aplicaciones, incluso algunas aún no imaginadas. Una vez introducido este tipo de tecnologías en la sociedad, comienzan a aparecer sistemas y servicios basados en tecnologías inalámbricas, mejorándose los procedimientos que tradicionalmente requerían una interacción directa con el ser humano y pueden ahora realizarse de forma distribuida por medio de sistemas gestores inteligentes.

Concretamente, desde hace algunos años, han comenzado a emerger las Redes de Sensores. Los sensores son fuentes de información tan variados como lo son las medidas que realizan. Los hay de temperatura, de luminosidad, de presión, de humedad, de velocidad, de aceleración, de presencia, de volumen y un sin fin de magnitudes que se nos ocurran. Si a estos sensores que nos reportan información valiosa para nuestras vidas, les añadimos la capacidad de comunicación inalámbrica y la posibilidad de formación de redes, obtenemos las Redes de Sensores Inalámbricas, que están teniendo un auge cada vez mayor debido principalmente a la multitud de aplicaciones que se están desarrollando, como aplicaciones de seguimiento, de seguridad, de salud, de gestión, etc.

Más aún, estamos alcanzando límites que pueden parecernos lejanos hoy en día, pero que muy pronto estaremos siendo parte de ellos. En computación, el «**Internet de las cosas**» se refiere a una red de objetos cotidianos interconectados. El concepto de Internet de las cosas se atribuye a Auto-ID

Center, fundado en 1999 y basado en el MIT. La idea es muy simple pero su aplicación es difícil. Si todas las latas, libros, zapatos o partes de un vehículo estuvieran equipados con dispositivos de identificación minúsculos, la vida cotidiana en nuestro planeta sufriría una transformación. Ya no existirían cosas fuera de stock o productos perdidos, porque nosotros sabríamos exactamente lo que se consume en el otro lado del planeta. El robo sería una cosa del pasado, sabríamos donde esta el producto en todo momento. Lo mismo se aplica a los paquetes perdidos.

Si todos los objetos de la vida cotidiana, desde el yogur a un avión, estuvieran equipados con etiquetas de radio, podrían ser identificados y gestionados por equipos de la misma manera que si lo fuesen por seres humanos. Con la próxima generación de aplicaciones de Internet (protocolo IPv6) se podría identificar todos los objetos, algo que no se puede hacer con IPv4, el sistema actualmente en uso, ya que la capacidad de manejo de direcciones es limitada, pero se resuelve perfectamente en IPv6, donde hay miles de millones de direcciones disponibles. Este sistema sería, por tanto capaz de identificar instantáneamente cualquier tipo de objeto.

"*El Internet de las cosas debe codificar de 50 a 100.000 millones de objetos y seguir el movimiento de estos. Notemos que todo ser humano está rodeado de aproximadamente unos 1.000 a 5.000 objetos.*"

Índice general

Índice de figuras

Capítulo 1

Redes de Sensores Inalámbricos

1.1. Introducción

Un sistema WSN (Wireless Sensor Network) de sensores inalámbricos es un red con numerosos dispositivos distribuidos espacialmente, que utilizan sensores para controlar diversas condiciones en distintos puntos, entre ellas la temperatura, el sonido, la vibración, la presión y movimiento o los contaminantes. Los dispositivos son unidades autónomas que constan de un microcontrolador, una fuente de energía (generalmente una batería, aunque los sensores autoalimentados están ganando terreno), un radiotransceptor y un elemento sensor y/o actuador.

Debido a las limitaciones de la vida de la batería, los nodos se construyen teniendo presente la conservación de la energía (eficiencia energética), y generalmente pasan mucho tiempo en modo "durmiente" (sleep) de bajo consumo de potencia. Los nodos auto-organizan sus redes en una forma ad hoc, en lugar de tener una topología de red previamente programada. Además, WSN tiene capacidad de autorrestauración (self-healing), es decir si se avería un nodo, la red encontrará nuevas rutas para encaminar los paquetes de datos. De esta forma, la red sobrevivirá en su conjunto, aunque haya nodos individuales que pierdan potencia o se destruyan.

Aunque es un tema de investigación controvertido, este punto de vista, más bien clásico, de WSN tiene pocas aplicaciones interesantes. Por ejemplo, algunos autores especializados en este campo señalan la detección de incendios forestales como una de las aplicaciones de WSN.

El MIT (Massachusetts Institute of Technology) identificó en Febrero de 2003 las diez tecnologías emergentes que cambiarán el mundo y las redes

de sensores inalámbricas aparecían en primer lugar. Las Redes de Sensores Inalámbricas (Wireless Sensor Networks, WSN) consisten en un conjunto de nodos de pequeño tamaño, de muy bajo consumo y capaces de una comunicación sin cables, interconectados entre sí a través de una red y a su vez conectados a un sistema central encargado de recopilar la información recogida por cada uno de los sensores.

Este novedoso tipo de redes se ha convertido en un campo de estudio que se encuentra en continuo crecimiento, ya que últimamente han surgido una serie de dispositivos que integran un procesador, una pequeña memoria, sensores y comunicación inalámbrica y todo ello, de ultra bajo consumo (como se verá más adelante en el presente trabajo, este tipo de dispositivos se denominan *motes*). Al estar dotados con un procesador o MCU (Micro-Controller Unit), estos nodos son capaces de realizar ciertas computaciones locales sobre los datos adquiridos, lo que permite una serie de ventajas tales como una reducción del tráfico a través de la red, ya que será procesada localmente, y consecuentemente una lógica descarga de trabajo del nodo o computador central.

Las redes de sensores con cables no son nuevas y sus funciones incluyen medir niveles de temperatura, líquido, humedad etc. Muchos sensores en fábricas o coches por ejemplo, tienen su propia red que se conecta con un ordenador o una caja de control a través de un cable y, al detectar una anomalía, envían un aviso a la caja de control.

La diferencia entre los sensores que todos conocemos y la nueva generación de redes de sensores inalámbricas es que estos últimos son inteligentes, es decir, capaces de poner en marcha una acción según la información que vayan acumulando, y no están limitados geográficamente y físicamente por un cable fijo.

Los nuevos avances en la fabricación de microchips de radio, nuevas formas de routers y nuevos programas informáticos relacionados con redes están logrando eliminar los cables de las redes de sensores, multiplicando así su

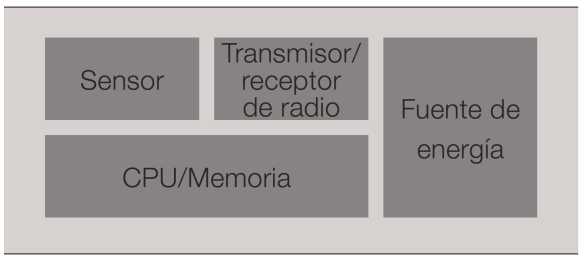


Figura 1.1: Ejemplo de la arquitectura de un nodo de una WSN.

Figura 1.2: Red de Sensores Inalámbricos y/o Autoalimentados.

potencial.

El ámbito de aplicación de este tipo de sistemas, como veremos, es muy amplio: monitorización de entornos naturales, aplicaciones para defensa, aplicaciones médicas en observación de pacientes, etc. El motivo del éxito de este tipo de redes de sensores se debe a sus especiales características físicas. A los nodos de las redes se les imponen unas *restricciones de consumo energético muy importantes*. El motivo de la imposición de estas restricciones es la necesidad de que los nodos sean capaces de operar, por sí mismos, durante períodos largos de tiempo, en lugares donde las fuentes de alimentación son si no inexistentes, de baja potencia. El *tamaño* es otra restricción que cada vez se hace más necesaria para la mayoría de las aplicaciones, de manera que los nodos que forman las redes de sensores sean cada vez de menor tamaño.

Desde el punto de vista del software, para la realización de las aplicaciones para redes de sensores, la Universidad de Berkeley e Intel han desarrollado una plataforma específica para este tipo de sistemas, que tiene en cuenta las restricciones de los nodos. En particular se ha desarrollado un sistema operativo, llamado **TinyOS**, cuya característica principal reside en que al ser *modular resulta ideal para instalarse en sistemas con restricciones de memoria.*

Se desarrolló también un lenguaje de programación, llamado **nesC**, de sintaxis muy parecida a C, basado en componentes, y a partir del cual se

rediseñó una primera versión de TinyOS de modo que actualmente está íntegramente implementado sobre nesC. Tanto nesC como TinyOS están basados en componentes e interfaces bidireccionales.

Además, actualmente, Berkeley e Intel han desarrollado diversas aplicaciones a modo de ejemplo, simuladores de ejecución, y varias universidades internacionales están dedicando esfuerzos al desarrollo de aplicaciones usando esta emergente tecnología. Existen otras empresas que son proveedores de esta tecnología, el mayor de estos es Crossbow Technology, que ha desarrollado redes de sensores a gran escala para su uso comercial.

Las últimas investigaciones apuntan hacia una eventual proliferación de redes de sensores inteligentes, redes que recogerán enormes cantidades de información hasta ahora no registrada que contribuirá de forma favorable al buen funcionamiento de fábricas, al cuidado de cultivos, a tareas domésticas, a la organización del trabajo y a la predicción de desastres naturales como los terremotos. En este sentido, la computación que penetra en todas las facetas de la vida diaria de los seres humanos está a punto de convertirse en realidad.

Si los avances tecnológicos en este campo siguen a la misma velocidad que lo han hecho en los últimos años, las redes de sensores inalámbricas revolucionarán la capacidad de interacción de los seres humanos con el mundo.

1.2. Historia

Las redes de sensores nacen, como viene siendo habitual en el ámbito tecnológico, de aplicaciones de carácter militar.

La primera de estas redes fue desarrollada por Estados Unidos durante la guerra fría y se trataba de una red de sensores acústicos desplegados en el fondo del mar cuya misión era desvelar la posición de los silenciosos submarinos soviéticos, el nombre de esta red era SOSUS (Sound Surveillance System). Paralelamente a ésta, también EE.UU. desplegó una red de radares aéreos a modo de sensores que han ido evolucionando hasta dar lugar a los famosos aviones AWACS, que no son más que sensores aéreos. SOSUS ha evolucionado hacia aplicaciones civiles como control sísmico y biológico, sin embargo AWACS sigue teniendo un papel activo en las campañas de guerra.

A partir de 1980, la DARPA comienza un programa focalizado en sensores denominado DSN (Distributed Sensor Networks), gracias a él se crearon sistemas operativos (Accent) y lenguajes de programación (SPLICE) orientados de forma específica a las redes de sensores, esto ha dado lugar a nuevos sistemas militares como CEC (Cooperative Engadgement Capability) consis-

tente en un grupo de radares que comparten toda su información obteniendo finalmente un mapa común con una mayor exactitud.

Estas primeras redes de sensores tan sólo destacaban por sus fines militares, aún no satisfacían algunos requisitos de gran importancia en este tipo de redes tales como la autonomía y el tamaño. Entrados en la década de los 90, una vez más DARPA lanza un nuevo programa enfocado hacia redes de sensores llamado SensIt, su objetivo viene a mejorar aspectos relacionados con la velocidad de adaptación de los sensores en ambientes cambiantes y en cómo hacer que la información que recogen los sensores sea fiable.

Ha sido a finales de los años 90 y principios de nuestro siglo cuando los sensores han empezado a tomar una mayor relevancia en el ámbito civil, decreciendo en tamaño e incrementando su autonomía. Compañías como Crossbow han desarrollado nodos sensores del tamaño de una moneda con la tecnología necesaria para cumplir su cometido funcionando con baterías que les hacen tener una autonomía razonable y una independencia inédita.

El futuro ya ha empezado a ser escrito por otra compañía llamada Dust Inc, compuesta por miembros del proyecto Smart Dust ubicado en Berkeley, que ha creado nodos de un tamaño inferior al de un guisante y que, debido a su minúsculo tamaño, podrán ser creadas múltiples nuevas aplicaciones.

Figura 1.3: Tamaño de los nodos Smart Dust (polvo inteligente).

1.3. Espectro de Frecuencia

Dentro del espectro electromagnético, que podemos ver en la Figura 1.4, se usan principalmente las bandas infrarrojos (IR) y de radiofrecuencia (RF) para transmitir señales de forma inalámbrica. Las comunicaciones por IR pueden alcanzar tasas de transferencia de hasta 4 Mbps o más, pero presentan el inconveniente de que sólo permiten las comunicaciones para pequeñas distancias y siempre que el transmisor y el receptor tengan línea directa de visión. Su uso se centra principalmente en las comunicaciones de periféricos electrónicos como teléfonos móviles. Por contra, las comunicaciones por radiofrecuencia a la vez que poseen altas tasas de transferencia, también permiten la comunicación entre dos puntos muy alejados y sin línea directa de visión.

Dentro del espectro de RF podemos encontrar la banda ISM (Industrial, Scientific and Medical). Esta es una parte del espectro electromagnético de propósito general, que puede ser usada sin necesidad de licencia siempre que se respeten unos ciertos límites de potencia emitida. La banda ISM fue diseñada por el sector de radiocomunicaciones de la Unión Internacional de Telecomunicaciones (ITU-R). El único requerimiento para desarrollar un producto que use la banda ISM es adaptarse a diversas normas especificadas por los organismos que regulan el espectro electromagnético en diversas zonas geográficas. En concreto, si nos encontramos en Europa tendremos que seguir las normas de la ETSI (European Telecommunications Standars Institute) y si nos encontramos en los Estados Unidos las de la FCC (Federal Communication Commision).

Los sistemas diseñados para trabajar en la banda ISM se caracterizan por un bajo consumo y, en principio, tasas de transmisión no muy altas. No obstante, en los últimos años se ha venido trabajando intensamente para aumentar dichas tasas de transmisión de datos. La mayor parte de los sistemas que trabajan en la banda ISM lo hacen bien a 2.4GHz o bien en la banda por debajo de 1GHz. Mientras la banda a 2.4GHz es universal, por debajo de 1GHz las frecuencias estandarizadas varían de un país a otro. Por ejemplo, mientras en Europa la frecuencia estándar por debajo de 1GHz es 868MHz, en Estados Unidos se usa aquella centrada en los 915MHz. En la Figura 1.5 podemos ver las frecuencias usadas en diferentes partes del globo. A la hora de elegir una determinada frecuencia de trabajo, los diseñadores deben tener muy en cuenta las ventajas y desventajas asociadas a dicha elección.

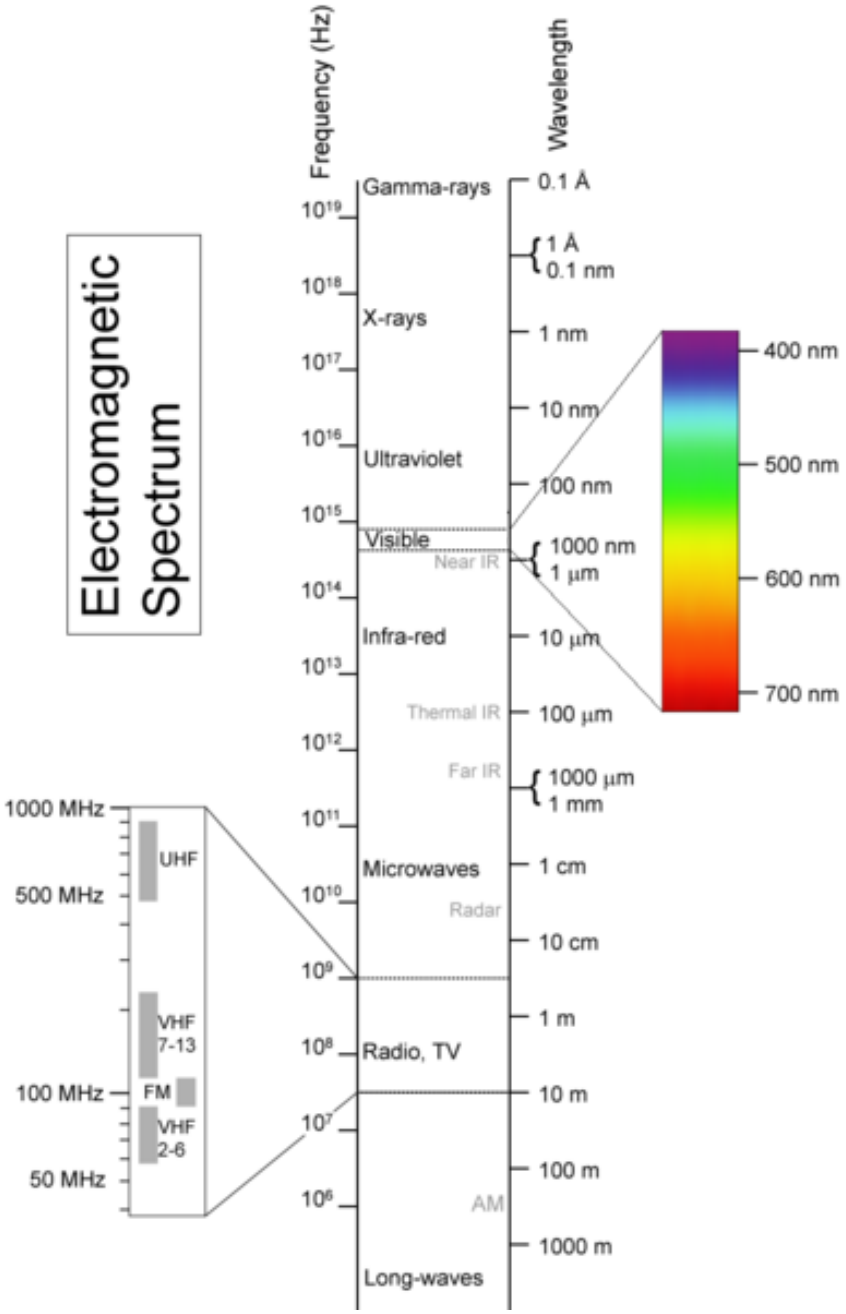


Figura 1.4: Espectro electromagnético.

En concreto:

- La banda de 2.4GHz es recomendable cuando la interoperabilidad es un requerimiento para nuestro dispositivo. Además, si necesitamos que dicho dispositivo funcione correctamente en distintas zonas geográficas, 2.4GHz es la mejor elección. Por otro lado, puesto que a esta frecuencia trabajan un gran número de tecnologías (Wi-Fi, Bluetooth, ZigBee, etc) e incluso algunos hornos microondas, los altos niveles de interferencia son un problema a superar. Por último, una frecuencia mayor implica que la señal será absorbida más fácilmente por el entorno y los objetos circundantes, con lo que el rango de alcance disminuirá respecto al caso de usar frecuencias menores.

  Además, la banda posee un ancho de banda más grande, permitiendo muchos canales separados y altas tasas de datos. También es posible utilizar ciclos de trabajo del 100 % y por último, es posible lograr obtener antenas más compactas que para frecuencias debajo de 1GHz.

- Usando el rango de frecuencias por debajo de 1GHz mejoramos algunas de las características que presentaban problemas para el caso de los 2.4GHz, en concreto aquellas relacionados con las interferencias y con el rango de alcance. No obstante, y como ya sabemos, trabajar con una frecuencia menor hace que disminuya el ciclo de trabajo y, en consecuencia, el ancho de banda. Además la interoperabilidad y la mo-

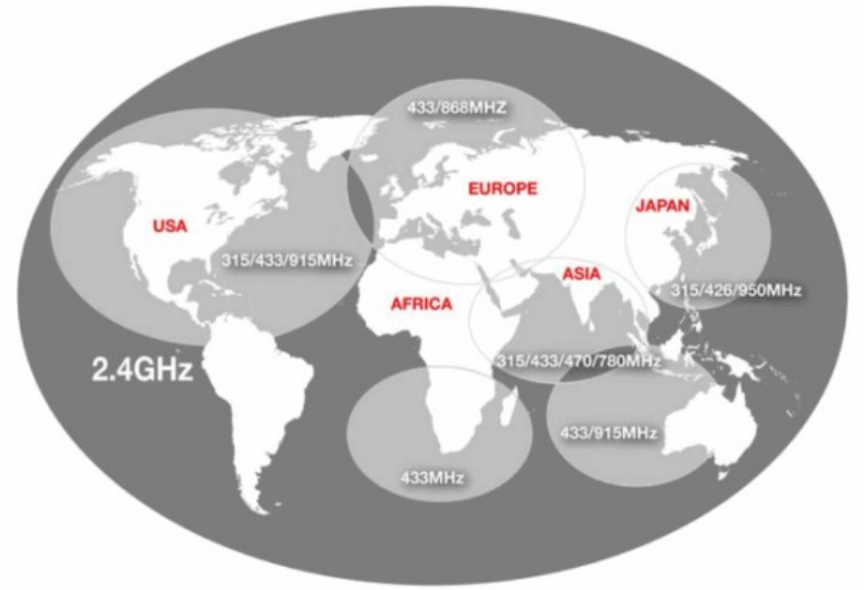


Figura 1.5: Bandas de frecuencia ISM/SRD.

Standard	Bluetooth	UWB	ZigBee	Wi-Fi
IEEE spec.	802.15.1	802.15.3a *	802.15.4	802.11a/b/g
Frequency band	2.4 GHz	3.1-10.6 GHz	868/915 MHz; 2.4 GHz	2.4 GHz; 5 GHz
Max signal rate	1 Mb/s	110 Mb/s	250 Kb/s	54 Mb/s
Nominal range	10 m	10 m	10 - 100 m	100 m
Nominal TX power	0 - 10 dBm	-41.3 dBm/MHz	(-25) - 0 dBm	15 - 20 dBm
Number of RF channels	79	(1-15)	1/10; 16	14 (2.4 GHz)
Channel bandwidth	1 MHz	500 MHz - 7.5 GHz	0.3/0.6 MHz; 2 MHz	22 MHz
Modulation type	GFSK	BPSK, QPSK	BPSK (+ ASK), O-QPSK	BPSK, QPSK COFDM, CCK, M-QAM
Spreading	FHSS	DS-UWB, MB-OFDM	DSSS	DSSS, CCK, OFDM
Coexistence mechanism	Adaptive freq. hopping	Adaptive freq. hopping	Dynamic freq. selection	Dynamic freq. selection, transmit power control (802.11h)
Basic cell	Piconet	Piconet	Star	BSS
Extension of the basic cell	Scatternet	Peer-to-peer	Cluster tree, Mesh	ESS
Max number of cell nodes	8	8	> 65000	2007
Encryption	E0 stream cipher	AES block cipher (CTR, counter mode)	AES block cipher (CTR, counter mode)	RC4 stream cipher (WEP), AES block cipher
Authentication	Shared secret	CBC-MAC (CCM)	CBC-MAC (ext. of CCM)	WPA2 (802.11i)
Data protection	16-bit CRC	32-bit CRC	16-bit CRC	32-bit CRC

Figura 1.6: Comparativa entre los protocolos Wi-Fi, Bluetooth, ZigBee y UWB.

vilidad geográfica de nuestro dispositivo también se ven ampliamente mermandas.

Por otra parte, este rango de frecuencias logran una mejor penetración a través del concreto y el acero, comparado con 2.4GHz. Además, se presentan en algunas regiones reestricciones respecto del ciclo de trabajo.

Para el caso de nuestro proyecto nos centraremos en dispositivos que trabajan en la banda de los 2.4GHz puesto que damos prioridad a la interoperabilidad, la movilidad geográfica y una tasa de bits aceptable.

En los últimos años hemos asistido al desarrollo de diversos estándares inalámbricos que operan en la banda ISM. En consecuencia, un gran número de productos basados en tecnología inalámbrica han visto la luz. Dichos estándares se diferencian unos de otros por su tasa de transferencia, rango de alcance, sus dominios de aplicación, técnicas de modulación usadas, etc. A día de hoy, los más usados y con mayor proyección de futuro son Wi-Fi, Bluetooth, ZigBee y Ultra-WideBand (UWB) (deberíamos también incluir a 6LoWPAN). En la Figura 1.6 podemos ver una comparativa de dichos estándares y en la Figura 1.7 vemos representado de forma aproximada su rango de alcance respecto a la tasa de transferencia.

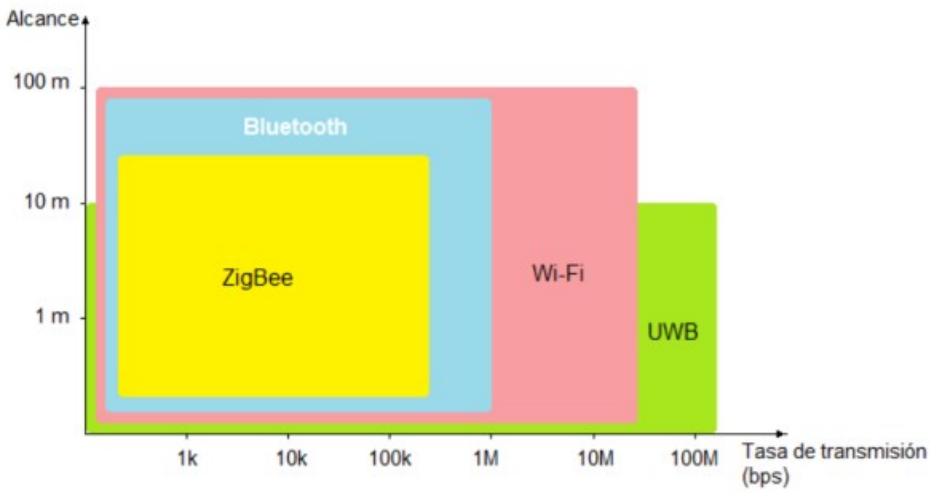


Figura 1.7: Rango respecto a tasa de transferencia para diferentes tecnologías inalámbricas.

1.4. Características de las WSN

Los recientes avances en microelectrónica, wireless y electrónica digital han permitido el desarrollo de nodos sensores de bajo costo, reducido tamaño, bajo consumo y que se comunican de forma inalámbrica.

El desarrollo de estos nodos sensores ha dado la posibilidad de crear redes basadas en cooperación de los nodos, con una notable mejora sobre redes de sensores tradicionales, que se suelen desplegar de dos modos:

- Sensores que se encuentran lejos del fenómeno, grandes y muy complejos para distinguir el objetivo del ruido del entorno.
- Muchos sensores con posición y topología cuidadosamente seleccionada. Transmiten datos de adquisición a nodos centrales que realizan la computación.

Como se ha comentario anteriormente, las WSN se componen de miles de dispositivos pequeños, autónomos, distribuidos geográficamente, llamados nodos sensores con capacidad de cómputo, almacenamiento y comunicación en una red conectada sin cables, e instalados alrededor de un fenómeno objeto para monitorearlo.

Una vez se produzcan eventos, toma de medidas o cualquier actividad programada con el fenómeno en cuestión los nodos enviarán información a través de la red, hasta llegar a un sistema central de control que recogerá los

datos y los evaluará, ejecutando las acciones pertinentes en comunicación con otros sistemas o en la propia red de sensores.

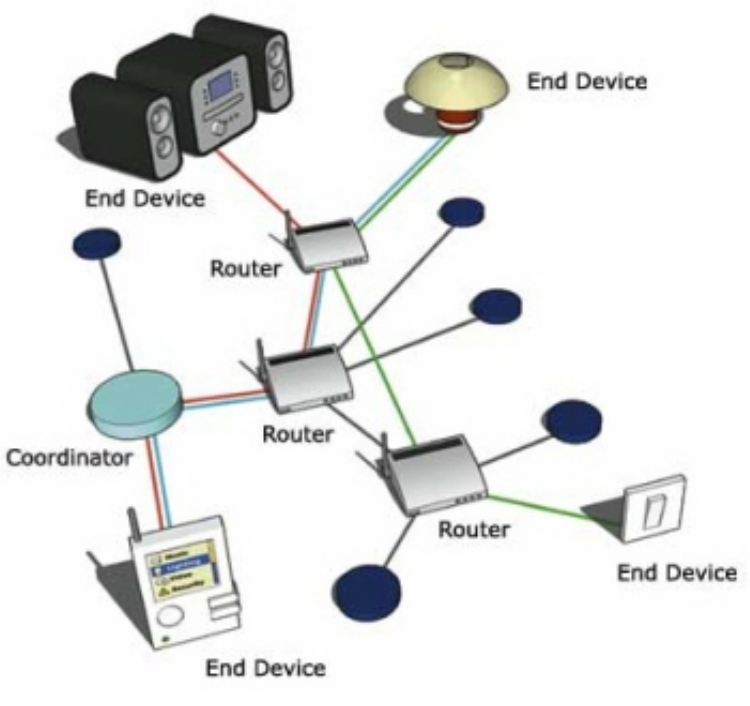


Figura 1.8: Elementos de una WSN típica.

Como se puede observar en la Figura 1.8, se puede establecer, por tanto, una serie de elementos que componen de forma general una WSN:

- **Sensores**: de distinta naturaleza y tecnología, toman del medio la información y la convierten en señales eléctricas.
- **Nodos sensores**: son los sensores inalámbricos, que toman los datos del sensor a través sus puertos, procesan la información y la envían a la estación base.
- **Gateways**: son elementos para la interconexión entre la red de sensores y una red, por ejemplo, TCP/IP.
- **Estaciones base/Router/Coordinador**: Recolector de datos basado en un PC o sistema integrado, que pueden realizar funciones de gestión sobre la red.
- **Red Inalámbrica**: típicamente basada en el estándar IEEE 802.15.4.

1.4.1. Características

Aunque la naturaleza de los nodos que emplean las redes pueda ser muy distinta y la misión a realizar pueda variar dependiendo del tipo de aplicación, se pueden identificar una serie de características comunes a todas ellas y que son las que principalmente las identifican:

- **Gran Escala**

  La cantidad de nodos que se despliega en una red puede llegar hasta los miles, pudiendo crecer su número a lo largo de la vida de la red. La red se va a componer de muchos sensores densamente desplegados en el lugar donde se produce el fenómeno y, por lo tanto, muy cerca de él. Hablamos de fenómeno y lo podemos relacionar con la naturaleza, pero podríamos asociarlo también a fenómenos sociales, etc.

- **Topología variable**

  La posición en que se coloca cada nodo puede ser arbitraria y normalmente es desconocida por los otros nodos. La localización no tiene porqué estar diseñada ni preestablecida, lo que va a permitir un despliegue aleatorio en terrenos inaccesibles u operaciones de alivio en desastres. Por otro lado, los algoritmos y protocolos de red deberán ser autoorganizativos.

- **Recursos limitados**

  Los sensores, a cambio de su bajo consumo de potencia, coste y pequeño tamaño disponen de recursos limitados. Los dispositivos actuales más usados, los mica2, cuentan con procesadores a 4 MHz, 4 Kbytes de RAM, 128 Kbytes de memoria de programa y 512 Kbytes de memoria flash para datos. Su radio permite trasmitir a una tasa de 38.4 KBaudios.

- **Cooperación de nodos sensores**

  Realizan operaciones simples antes de transmitir los datos, lo que se denomina un procesamiento parcial o local.

- **Comunicación**

  Los nodos sensores usan comunicación por difusión (broadcast[1]) y debido a que están densamente desplegados, los vecinos están muy cerca unos de otros y la comunicación multihop (salto múltiple de uno a otro) consigue un menor consumo de potencia que la comunicación single hop (salto simple). Además, los niveles de transmisión de potencia se mantienen muy bajos y existen menos problemas de propagación que en comunicaciones inalámbricas de larga distancia.

[1] Como el número de nodos en una red puede llegar a ser muy grande, no se utilizan identificadores globales. De ahí el tipo de comunicación broadcast.

- **Funcionamiento autónomo**

  Puede que no se acceda físicamente a los nodos por un largo periodo de tiempo. Incluso tal vez esto nunca ocurra. Durante el tiempo en el que el nodo permanece sin ser atendido puede que sus baterías se agoten o su funcionamiento deje de ser el esperado.

- **Heterogeneidad**

  Los nodos sensores no tienen por qué ser todos iguales, sino que pueden estar midiendo aspectos diferentes de un mismo fenómeno. Además, estos nodos pueden ser reemplazados y sustituidos por otros que aunque realicen las mismas funciones son diferentes.

1.4.2. Requisitos

Para que una red pueda funcionar de acuerdo con las anteriores características surgen una serie de retos que la aplicación debe resolver. Estos, que se detallan a continuación, son los requisitos no funcionales del sistema:

- **Eficiencia energética**

  Es uno de los asuntos más importantes en redes de sensores. Cuanto más se consiga bajar el consumo de un nodo mayor será el tiempo durante el cual pueda operar y, por tanto, mayor tiempo de vida tendrá la red. La aplicación tiene la capacidad de bajar este consumo de potencia restringiendo el uso de la CPU y la radio. Esto se consigue desactivándolos cuando no se utilizan (activando el modo de bajo consumo) y, sobre todo, disminuyendo el número de mensajes que generan y retransmiten los nodos.

- **Autoorganización**

  Los nodos desplegados deben formar una topología que permita establecer rutas por las que mandar los datos que han obtenido. Los nodos necesitan conocer su lugar en esta topología, pero resulta inviable asignarlo manualmente a cada uno, debido al gran número de estos. Es fundamental, por tanto, que los nodos sean capaces de formar la topología deseada sin ayuda del exterior de la red. Este proceso no sólo debe ejecutarse cuando la red comienza su funcionamiento, sino que debe permitir que en cada momento la red se adapte a los cambios que pueda haber en ella.

- **Escalabilidad**

  Puesto que las aplicaciones van creciendo con el tiempo y el despliegue de la red es progresivo, es necesario que la solución elegida para la red permita su crecimiento. No sólo es necesario que la red funcione

correctamente con el número de nodos con que inicialmente se contaba, sino que también debe permitir aumentar ese número sin que las prestaciones de la red caigan drásticamente.

- **Tolerancia a Fallos**

  Los sensores son dispositivos propensos a fallar. Los fallos pueden deberse a múltiples causas, pueden venir a raíz del estado de su batería, de un error de programación, de condiciones ambientales, del estado de la red, etc. Una de las razones de esta probabilidad de fallo radica precisamente en el bajo coste de los sensores. En cualquier caso, se deben minimizar las consecuencias de ese fallo. Por todos los medios se debe evitar que un fallo en un nodo individual provoque el mal funcionamiento del conjunto de la red.

- **Tiempo Real**

  Los datos llegan a su destino con cierto retraso. Pero algunos datos deben entregarse dentro de un intervalo de tiempo conocido. Pasado este dejan de ser válidos, como puede pasar con datos que impliquen una reacción inmediata del sistema, o se pueden originar problemas serios como ocurriría si se ignora una alarma crítica. En caso de que una aplicación tenga estas restricciones deben tomarse las medidas que garanticen la llegada a tiempo de los datos.

- **Seguridad**

  Las comunicaciones wireless viajan por un medio fácilmente accesible a personas ajenas a la red de sensores. Esto implica un riesgo potencial para los datos recolectados y para el funcionamiento de la red. Se deben establecer mecanismos que permitan tanto proteger los datos de estos intrusos, como protegerse de los datos que estos puedan inyectar en la red.

Según la aplicación que se diseñe algunos de los anteriores requisitos cobran mayor importancia. Como ejemplo podemos considerar una aplicación que controle el comportamiento de animales salvajes dentro de un parque natural. Para determinar su localización, cada uno de estos animales llevaría sujeto un pequeño sensor. En esta situación la capacidad de autoorganización cobraría gran importancia. Sin embargo, si pensamos en una red con nodos inmóviles, como los usados en la red domótica de una oficina, este mismo atributo sería de menor importancia.

Es necesario encontrar el peso que cada uno de estos requisitos tiene en el diseño de la red, pues normalmente unos requisitos van en detrimento de otros. Por ejemplo, dotar a una red de propiedades de tiempo real podría implicar aumentar la frecuencia con la que se mandan mensajes con datos,

lo cual repercutiría en un mayor consumo de potencia y un menor tiempo de vida de los nodos.

Esto lleva a buscar, para cada aplicación, un compromiso entre los requisitos anteriores que permita lograr un funcionamiento de la red adecuado para la misión que debe realizar.

1.5. Protocolos de la WSN

A continuación vamos a ver por qué son diferentes las WSN de las redes tradicionales y *por qué existen algoritmos y protocolos que no se ajustan a las redes de sensores*, ya que se encuentran diferencias fundamentales en los principales objetivos de ambas redes.

Las WSN utilizan una comunicación inalámbrica y, obviamente, son diferentes de las redes cableadas. También son diferentes de las tradicionales redes inalámbricas como las redes celulares, Bluetooth o las móviles ad hoc (MANETS). En estas redes, el *objetivo es optimizar el rendimiento y el retardo*. Aunque las MANETS comparten las características de desarrollo ad hoc y autoconfiguración de los nodos, *el consumo de potencia no es una prioridad*. Bluetooth también trata los mismos problemas de limitación de potencia pero el grado de bajo consumo de potencia que se requiere en las WSN es mucho mayor. Además, los nodos sensores son frecuentemente expuestos a extremas condiciones ambientales, haciéndolos propensos a frecuentes fallos en los nodos. Esto conlleva unas restricciones estrictas en las WSN, no como en las otras redes.

En la Figura **??** se una pila de protocolos que dispone de 5 capas y 3 planos que son comunes a cada una de las capas. Cada una de las capas del protocolo responde a características similares a la tradicional pila de protocolos de red, de ahí que reciban el mismo nombre.

El plano de gestión de energía controla el uso de energía por parte del nodo sensor, por ejemplo, apagando el nodo después de recibir un mensaje de uno de sus nodos vecinos, evitando así la recepción de mensajes duplicados, o comunicando al resto de nodos que no participará en el enrutamiento de mensajes si le queda poca batería. El plano de gestión de la movilidad controla el movimiento del sensor, manteniendo conocimiento sobre cuáles son los nodos vecinos. Por último, el plano de gestión de tareas se encarga de controlar y planificar las tareas de sensorización del nodo en una determinada zona, ya que es posible, que dependiendo de los niveles de batería, unos nodos trabajen más que otros.

1.5.1. Capa Física

La capa física, como en otros casos, define la selección de la frecuencia, la generación de la frecuencia de la portadora, la detección de señales, la modulación, etc.

Para las redes de sensores las capas físicas útiles incluyen: IEEE 802.15.4, Bluetooth, IEEE 802.11, Ethernet, IrDA (InfraRed Data Association), RF (Radio Frequency), Powerline Networking, Wireless USB, entre otras. Algunas de las capas físicas mencionadas como IrDA o Bluetooth usan propietariamente como capas físicas a otras más sencillas como infrarrojo o RF respectivamente.

IEEE 802.15.4 se usa generalmente con Zigbee en las capas superiores. Bluetooth, IrDA, Wireless USB son fácilmente utilizables por la interfaz serial que presentan a las aplicaciones. El par trenzado, ethernet, RF y el powerline networking se usan en tecnologías de redes de sensores como KNX, X10, Insteon.

Bluetooth usa la banda ISM de 2.4GHz, para evitar interferencias usa FHSS (Frequency Hopping Spread Spectrum), soporta dos modos de enlaces: ACL (Asynchronous ConnectionLess) para transmisión de datos y SCO (Synchronous Connection Oriented) para transmisión de audio y voz. En ACL se tiene velocidades de 721/57.6Kbps. Y en SCO 3 canales de 64Kbps/dispositivo.

IEEE 802.15.4, define 4 rangos de frecuencia de operación, y una técnica de modulación para cada rango de frecuencia, para evitar interferencias usa PSSS (Parallel Sequence Spread Spectrum) para el tercer caso y DSSS (Direct Sequence Spread Spectrum) para las demás capas físicas.

PHY (MHz)	Frequency band (MHz)	Spreading parameters		Data parameters		
		Chip rate (kchip/s)	Modulation	Bit rate (kb/s)	Symbol rate (ksymbol/s)	Symbols
868/915	868–868.6	300	BPSK	20	20	Binary
	902–928	600	BPSK	40	40	Binary
868/915 (optional)	868–868.6	400	ASK	250	12.5	20-bit PSSS
	902–928	1600	ASK	250	50	5-bit PSSS
868/915 (optional)	868–868.6	400	O-QPSK	100	25	16-ary Orthogonal
	902–928	1000	O-QPSK	250	62.5	16-ary Orthogonal
2450	2400–2483.5	2000	O-QPSK	250	62.5	16-ary Orthogonal

Figura 1.9: Capas físicas de IEEE 802.15.4.

1.5.2. Capa de Enlace de Datos

Como en otros casos, la capa de enlace de datos es responsable del multiplexado del flujo de datos, la detección de tramas de datos, acceso al medio y control de errores; asegurando conexiones confiables punto a punto y punto a multipunto.

1.5.2.1. MAC (Medium Access Control)

En las redes de sensores, el protocolo de control de acceso al medio debe alcanzar dos objetivos:

- Establecer una infraestructura de red; que mediante la creación de enlaces de comunicación permiten realizar las transferencias de datos en redes de sensores pequeñas y muy grandes.
- Compartir los recursos de comunicación de manera justa y eficiente entre los nodos de la red de sensores.

A continuación se describe brevemente a los protocolos de control de acceso al medio para redes de sensores más representativos:

- **Algoritmos SMACS** (Self-Organizing Medium Access Control for Sensor Networks) y EAR (Eavesdrop and Register)

  SMACS es un protocolo de infraestructura distribuida que permite a los nodos descubrir a sus vecinos y programar la transmisión/recepción sin la necesidad de nodos maestros locales o globales.

  Se combina el descubrimiento de vecinos con la asignación de fases de los canales de manera que en el descubrimiento de vecinos ya se establece lazos de comunicación entre los nodos.

  En SMACS, la conservación de energía se consigue al usar tiempos aleatorios para despertar durante la fase de conexión y apagando la radio en los periodos libres de transmisión.

  El algoritmo EAR trata de ofrecer un servicio continuo a los nodos sensores en condiciones estacionarias y de movilidad. Los nodos móviles asumen control total del proceso de conexión y también deciden cuando descartar conexiones, reduciendo la sobrecarga de mensajes.

  EAR es transparente a SMACS y le otorga la funcionalidad de *manejar nodos móviles*, ya que SMACS funciona perfectamente cuando *los nodos se encuentran estacionarios.*

- **TDMA/FDMA**

  Es un esquema de control de acceso al medio centralizado, y se considera el efecto que produciría una capa física no ideal. Asume que el nodo sensor tiene únicamente capacidad de transmitir a los nodos más cercanos (distancias menores a 10m).

  El esquema TDMA, se dedica a usar todo el ancho de banda disponible en un solo nodo, mientras que el esquema FDMA, permite usar una cantidad mínima de ancho de banda por cada nodo.

  Para determinar el esquema TDMA-FDMA a usar, se busca el mínimo de consumo energético. Así, si el transmisor consume más potencia, se usa mayormente el esquema TDMA, mientras que si el receptor es quien consume más potencia, se usa predominantemente el esquema FDMA.

- **CSMA/CA** (Carrier Sense Multiple Access with Collision Avoidance)

  CSMA/CA está basado en la detección de canales físicos y virtuales. Si un nodo está listo para transmitir, solo lo hace en el caso de que detecte que el canal físico está inactivo. Una vez iniciada la transmisión, ya no se realiza detección del canal. En el caso de que el canal esté activo cuando se desea realizar la transmisión, el nodo espera hasta la inactividad del canal. Si se da el caso de colisión, se realiza el procedimiento de backoff para los respectivos reintentos.

  En el periodo de espera para retransmisión se puede realizar ahorro energético por desactivación de transceivers y otros elementos implicados en las transmisiones de datos.

Existen otros métodos de control de acceso al medio como Wins, Picoradio, 1-DSMA (Singly Persistent Data Sensing for Multiple Access) entre otros.

1.5.2.2. Modos de operación de bajo consumo

El modo de conservación energético más evidente a usar es el apagar el transmisor/receptor cuando no se requiere sus funciones. Puede haber modos de operación de ahorro energético dependiendo de los estados del microprocesador, de la memoria, del convertidor analógico/digital (A/D) y del transmisor/receptor.

La operación de los modos de ahorro energético puede optimizarse basándose en umbrales obtenidos de los tiempos de transición entre modos y el consumo individual de los modos involucrados.

1.5.2.3. Control de errores

Una de las principales funciones de la capa de enlace de datos es el *control de errores* sobre los datos transmitidos. En las redes de sensores existen aplicaciones donde el control de errores es crítico, entre ellas el seguimiento de móviles y monitoreo de maquinaria industriales.

El control de errores se puede clasificar básicamente en dos grupos: FEC (Forward Error Correction) y ARQ (Automatic Repeat Request).

ARQ es ineficiente para las redes de sensores inalámbricos por utilizar retransmisiones para la recuperación de los datos perdidos, las cuales representan costos adicionales en el uso de la energía.

FEC por su parte implica complejidad en la decodificación de los datos, requiriendo recursos de procesamiento considerables en los nodos sensores.

Debido a las restricciones de las redes de sensores inalámbricos, no es energéticamente eficiente aumentar la potencia de transmisión, ni retransmitir los datos así que se debe usar ***FEC optimizado*** para este tipo de redes, lo que implica que *se debe lograr que la energía ahorrada de las retransmisiones o aumento de potencia sea menor que la que se gasta en el procesamiento requerido para la reconstrucción de los datos.*

Este tipo de esquemas FEC para las redes de sensores inalámbricos, aún se encuentra en estudio.

1.5.3. Capa de Red

La capa de red en las redes de sensores inalámbricos, provee mecanismos y procedimientos funcionales para intercambiar unidades de datos de servicios de red entre dos entidades de transporte sobre una conexión de red.

Por eficiencia, esta capa debe tener en cuenta los siguientes aspectos:

- Ahorro energético.
- Direccionamiento basado en atributos.
- Sumarización de datos solo cuando no se ocasione pérdida significativa de información.
- Compatibilidad para fácil acoplamiento con otras tecnologías.

Los *algoritmos de enrutamiento* usados por las redes de sensores inalámbricos, incluyen a los siguientes.

1.5.3.1. Inundación (*Flooding*)

Un nodo que recibe datos o algún paquete, lo repite en difusión (dominio de broadcast) hasta llegar al número máximo de saltos permitido ó hasta llegar al destino.

No se requiere de costosos mantenimientos de la topología ni complejos algoritmos de descubrimientos de rutas. La sencillez de este algoritmo implica las siguientes deficiencias:

- Implosión (Implosion): Cuando a un mismo nodo le llega mensajes duplicados.
- Traslape (Overlap): Cuando dos nodos están observando la misma región, puede darse el caso que sensen el mismo estímulo al mismo tiempo, entregándose mensajes duplicados.
- No consideración de recursos (Resource blindness): No se toma en cuenta los recursos energéticos disponibles.

1.5.3.2. Murmuración (*Gossiping*)

Es una derivación del algoritmo de inundación. La murmuración no difunde los paquetes entrantes, sino que los envía a un vecino elegido aleatoriamente. El vecino que recibió el mensaje elije aleatoriamente a otro vecino y le envía el paquete.

Se logra vencer de alguna manera el problema de implosión pero tiene retardos altos en la propagación del mensaje a toda la red.

1.5.3.3. SPIN (*Sensor Protocols for Information via Negotiation*)

La familia de protocolos SPIN se desarrolló bajo dos ideas básicas: los nodos sensores operen más eficientemente y se conserve energía enviando descripciones en lugar de enviar todos los datos a la vez.

SPIN supera las deficiencias del protocolo de inundación mediante negociación y adaptación a la disponibilidad de recursos.

SPIN tiene tres tipos de datos: ADV (ADVising), una pequeña descripción de los datos disponibles que se envía antes de la transmisión de los mismos; REQ (REQuest), un mensaje enviado por los nodos interesados en los datos que se publicaron en el mensaje ADV; DATA, donde se envían los mensajes a los nodos sensores que así lo solicitaron mediante el mensaje REQ.

1.5.3.4. SAR (*Sequential Assignment Routing*)

El algoritmo SAR usa a SMACS y EAR. SAR crea múltiples árboles teniendo como raíz al nodo recolector de datos. Los árboles crecen hacia los nodos sensores, evitando a aquellos que ofrecen muy baja calidad de servicio (alta latencia, baja velocidad efectiva) y/o bajas reservas energéticas. Para ello se estiman dos parámetros:

- Recursos energéticos: se estiman por el número de paquetes que el nodo sensor puede enviar cuando este tiene el uso exclusivo de la ruta.
- Métrica aditiva de calidad de servicio: un valor alto de este parámetro significa baja calidad de servicio en la ruta.

También se considera la prioridad de los paquetes. Todas estas variables permiten la creación de árboles con la pertenencia de un nodo a varios árboles.

1.5.3.5. LEACH (*Low Energy Adaptive Clustering Hierarchy*)

LEACH es un protocolo basado en agrupación que minimiza la disipación energética en las redes de sensores. El propósito de LEACH es seleccionar aleatoriamente nodos sensores como directores de grupo, de tal manera que la alta disipación energética de la comunicación con la estación base se disperse a todos los nodos sensores de la red.

LEACH tiene dos fases, la fase de establecimiento y la fase de estabilización, siendo esta última la de mayor duración para reducir el *overhead*.

En la fase de establecimiento, un nodo sensor elige un número aleatorio entre 0 y 1, y si este número es menor que un umbral, se elige como director de grupo.

Luego de seleccionarse los directores de grupo, ellos notifican a todos los nodos de la red que ha recibido esta designación. Recibida la notificación, los nodos deciden el grupo al que quieren pertenecer basándose en el nivel de señal con que recibieron la notificación de los directores de grupo, y luego informan a los directores de grupo que serán parte del conjunto correspondiente. Acto seguido, los directores asignan tiempos de transmisión de datos a los nodos en un esquema similar a TDMA.

En la fase de estabilización, los nodos pueden sensar y transmitir los datos a los directores de grupo, quienes reunirán cierta cantidad de datos para transmitirlos al recolector. Pasado algún tiempo, los nodos entran nuevamente en la etapa de establecimiento, y se someterán a otro proceso de evaluación para elegir los directores de grupo.

1.5.3.6. Difusión dirigida

La difusión dirigida esta basada en el enrutado por características de los datos (data centric routing) combinados con un dato de interés.

La diseminación de datos por difusión dirigida se propone cuando el recolector envía un interés a todos los sensores, que es la descripción de la tarea. Los descriptores de la tarea son nombrados por asignación de pares de valores de atributos que describen la tarea. Cada nodo sensor almacena el interés en su caché. El valor almacenado contiene un campo de impresión de tiempo (timestamp) y varios campos de sentido de transmisión. Debido a que el interés se propaga a través de la red de sensores, los sentidos de transmisión se establecen desde la fuente hacia el recolector.

Cuando la fuente tiene datos para el interés publicado, envía los datos a través del camino de sentido de transmisión de ese interés.

1.5.3.7. Enrutado por rumor

El enrutado por rumor esta diseñado para situaciones en las que el enrutado por criterio geográfico no se aplica debido a la ausencia de un sistema de coordinación o cuando el fenómeno de interés no esta geográficamente relacionado.

Se usa agentes de larga vida que crean rutas hacia los eventos. Cuando cruza caminos largos y que no encuentran el evento, se ajusta el algoritmo que crea las rutas. En el caso de encontrarse con rutas cortas, se optimizan las rutas en la red actualizando las tablas de enrutamiento para reflejar la ruta más eficiente.

1.5.3.8. DSR (*Dynamic Source Routing*)

Es un protocolo de enrutamiento para redes inalámbricas completamente interconectadas (topología árbol + estrella). Similar a AODV (Ad Hoc On-Demand Distance Vector Routing) mencionado en secciones anteriores, crea las rutas únicamente bajo demanda de comunicación de algún host. Usa source routing en lugar de tablas de enrutamiento en los dispositivos intermedios.

La determinación de las rutas requiere la acumulación de las direcciones de cada dispositivo entre el origen y destino de la comunicación. Lo que implica un alto overhead, y paquetes grandes debido a que contienen la dirección de todos los dispositivos a través de los cuales pasará.

Contempla dos procedimientos principales que son: el descubrimiento y el mantenimiento de las rutas. El descubrimiento de rutas se inicia con un paquete Route Request, tras lo cual se realiza el intercambio de información hasta la conclusión de este procedimiento, y finaliza el procedimiento con un mensaje Route Reply generado cuando se ha alcanzado el destino final.

El procedimiento de mantenimiento de rutas se inicia cuando existe una transmisión fallida con una ruta conocida, generándose un mensaje Route Error, tras lo cual se volverá a usar el procedimiento Route Discovery para determinar una nueva ruta.

1.5.3.9. DSDV (*Destination-Sequenced Distance Vector routing*)

Es un esquema de enrutamiento basado en tablas para redes móviles ad hoc basado en el algoritmo Bellman-Ford.

Los registros de la tabla de enrutamiento contienen números de secuencia generados por el destino de la comunicación, y se usa para discriminar las rutas más actuales.

Las desventajas de DSDV incluyen: requerimiento de continua actualización de tablas de enrutamiento, mal desempeño con redes grandes, y no se recomienda su uso en redes altamente dinámicas.

1.5.3.10. AODV (*Ad Hoc On-Demand Distance Vector Routing*)

AODV se describe en el RFC-3561 y fue diseñado para utilizarse en redes inalámbricas Ad Hoc con topología hibrida árbol-estrella.

Es capaz de realizar enrutado unicast y multicast. Se caracteriza entre los protocolos reactivos, lo que implica que solo establece una ruta hacia algún destino solo si existe necesidad de crear esa ruta.

Cuando un nodo necesita establecer una conexión, este difunde una petición de conexión. Los demás nodos AODV repiten el mensaje, y almacenan una ruta temporal para poder responder al nodo original.

Cuando se llega a algún nodo que contenga una ruta hacia el nodo destino, se empieza con la devolución de la misma hacia el nodo solicitante, para que este empiece a usar la ruta con menor número de saltos y recicle las entradas de enrutamiento no utilizadas.

Cuando algún enlace falla se genera un mensaje de error y el proceso de descubrimiento de rutas se repite.

El mantenimiento de las rutas se realiza básicamente con la ayuda de los números de secuencia, que aseguran que la información es más actualizada que la existente.

Entre las desventajas de AODV se tiene: requiere más tiempo que otros protocolos de enrutamiento inalámbricos para establecer conexiones, además este proceso requiere más tráfico que sus similares.

1.5.3.11. ZRP (*Zone Routing Protocol*)

Es a la vez un protocolo proactivo y reactivo. ZRP divide su red en diferentes zonas, a las que se conoce como nodes local neighbourhood, un nodo puede pertenecer a diferentes zonas de diferentes tamaños. Este tamaño está definido por el radio de longitud, donde el perímetro de la zona está dado por el número de saltos.

Para la comunicación en la misma zona se usa el proactivo IARP (Intrazone Routing Protocol), y para la comunicación entre zonas se usa el reactivo IERP (Interzone Routing Protocol). Para el paso de mensajes Route Request se usa el protocolo BRP (Broadcast Resolution Protocol).

Una de las ventajas más notables de ZRP es la reducción de overhead de control tanto para sus operaciones proactivas como para las que son bajo demanda. La desventaja que se cita es cierto nivel de latencia para el descubrimiento de nuevas rutas.

1.5.4. Capa de Transporte

En general los objetivos de la capa de transporte son:

- Proveer *comunicación transparente entre la capa de aplicación y la capa de red* usando la multiplexación/demultiplexación de aplicaciones en la capa de red.
- Proveer un servicio de entrega de datos entre la fuente y el recolector con un *mecanismo de control de errores* acorde a la confiabilidad que la capa aplicación demande.
- Regular la cantidad de tráfico que pasará a través de la red mediante *mecanismos de control de flujo y de control de congestión.*

Para cumplir con las restricciones de las redes de sensores inalámbricos, las funcionalidades de la capa de transporte se ven afectadas por factores

como las limitaciones en hardware, procesamiento y en energía, y por tal razón ***no se pueden emplear mecanismos como las conexiones extremo a extremo, mecanismos de control basados en retransmisión y basados en ventanas entre otros*** ya que con las características de las redes de sensores inalámbricos no serán más que un desperdicio de recursos.

El flujo de datos más importante en la red de sensores tiene lugar en la transmisión de información desde los nodos fuente hasta el recolector, coordinador o access point. El sentido reverso en el flujo de la información se encarga de llevar información originada en el recolector, coordinador o access point y que esta dirigida a los nodos sensores, estos datos pueden ser comandos, peticiones, programación, etc.

1.5.4.1. ESRT (*Event-to-Sink Reliable Transport Protocol*)

En la capa de transporte de las redes de sensores inalámbricos no se habla de entrega confiable basada en paquetes, si no de comunicación confiable desde un evento hacia el recolector.

La transmisión TCP extremo a extremo no es factible en las Redes de Sensores Inalámbricos por que esta basado en las confirmaciones y retransmisiones extremo a extremo, que son gastos elevados de energía.

ESRT, basado en la noción de un evento en un radio determinado facilita la detección del evento por parte del recolector y mejora la confiabilidad dependiendo de la densidad de nodos sensores en el radio del evento.

1.5.4.2. Transporte recolector a sensores

Este tipo de flujo de datos contiene principalmente datos emitidos por el recolector con propósitos operacionales y de aplicaciones específicas. Estos datos que requieren una confiabilidad al 100 % incluyen: binarios del sistema operativo, archivos de configuración de programación y cambio de tarea; y peticiones y comandos de aplicaciones específicas.

Debido a la confiabilidad requerida en este caso si es necesario el uso de mecanismos de retransmisión y confirmaciones, sin comprometer un uso excesivo de recursos. La solución aplicada a esta problemática es el usar retransmisiones locales en lugar de retransmisiones extremo a extremo y únicamente confirmación de los paquetes mal recibidos en lugar de confirmar todos los paquetes correctamente recibidos.

Se puede utilizar antenas de mayor alcance en el recolector y de esta manera reducir el número de retransmisiones reduciendo el gasto de energía

en los nodos.

Una solución a nivel lógico es el uso del mecanismo PSFQ (*Pump slowly, fetch quickly*) que se basa en la inyección lenta de datos en la red para la creación de un pequeño caché de datos en los nodos sensores y una recuperación agresiva salto a salto en caso de pérdida de paquetes.

1.5.5. Sincronización

La sincronización se vuelve importante para lograr eficiencia energética, bajo costo y pequeñez de los equipos.

Los siguientes son algunos de los factores que influencian la sincronización temporal en sistemas más amplios y que también afectan de igual manera a las redes de sensores inalámbricos:

- **Fallas del clock**: saltos repentinos en el conteo del tiempo provocando variaciones en frecuencia y saltos de tiempo.
- **Retardo asimétrico**: debido a que en la comunicación inalámbrica de los nodos sensores se pueden tener diferentes retardos en el trayecto de ida y el de regreso de la información.
- **Temperatura**: debido al despliegue aleatorio de los nodos sensores en varios lugares del terreno, la variación de temperatura en diferentes instantes pueden causar aceleración o retardo del clock interno de los nodos sensores.
- **Ruido en frecuencia**: por lo general este tipo de ruido se debe a la inestabilidad de la señal de clock proporcionada por un cristal.
- **Ruido de fase**: algunos de los causantes del ruido de fase pueden ser la fluctuación de acceso en una interfaz a nivel de hardware, la variación de respuestas del sistema operante ante interrupciones, el jitter en el retardo de la red, además se puede deber a la técnica de acceso al medio y retardos por encolado de los datos.

Las técnicas de sincronización deben superar estos obstáculos, y también ser concientes de que las baterías de los nodos sensores son limitadas.

Entre estas técnicas se tiene:

- **NTP (Network Time Protocol)**: se basa en servidores temporales que deben ser a la vez legibles desde la red, robustos ante fallas y altamente precisos. La desventaja del uso de este protocolo es que algunos nodos para llegar a los servidores temporales necesitan pasar a través de otros nodos, mismos que pueden fallar.

- **EBS (Refered Broadcast Synchronization)**: básicamente el tiempo se traslada salto a salto desde el inicio o fuente de la difusión hasta el fin de la red. Dado que tras múltiples saltos se puede tener variaciones diversas, luego de la difusión se realiza comunicación entre nodos para transmitir offsets estimados.
- **TDP (Time Diffusion Protocol)**: permite comunicar el tiempo en la red con cierta tolerancia, misma que se puede ajustar dependiendo del propósito de la red de sensores inalámbricos. El protocolo de difusión de tiempo se autoconfigura eligiendo nodos maestros para sincronizar la red de sensores. Los nodos maestros se eligen tomando en cuenta los requerimientos energéticos y la precisión de los relojes.

1.5.6. Localización

Dado que los nodos sensores se dispersan aleatoriamente, necesitan saber su ubicación y la de los demás nodos para constituirse en una fuente confiable de información para los usuarios.

Un protocolo de localización debe ser:

- Flexible en cualquier terreno.
- Robusto ante fallas de los nodos.
- Menos sensible al ruido en la medición.
- Bajo en error en la estimación de la ubicación.

Existen dos técnicas de localización que cumplen con estos requisitos:

- ***La localización basada en beacon***, básicamente se compone de estimación y alineación; en la fase de alineación cada nodo determina la alineación o dirección en la que se encuentran sus vecinos. Entonces, la fase de estimación permite a los nodos vecinos que no conocen su ubicación usar la alineación estimada en la fase anterior y la ubicación conocida de los beacons para estimar sus localizaciones.
- ***La técnica basada en ubicación relativa***, usa PLF (Perceptive Localization Framework), donde un nodo es capaz de detectar y rastrear la ubicación de un nodo vecino usando una técnica de estimación conjunta y algún criterio aplicado al grupo de sensores. No se necesita una unidad central que difunda un beacon para determinar la localización de los nodos.

1.5.7. Capa de Aplicación

1.5.7.1. SMP *Sensor Management Protocol*

La función principal de SMP es hacer transparentes al hardware y software de las capas inferiores con respecto a las aplicaciones de administración de la red de sensores.

SMP es un protocolo de administración que provee las operaciones de software requeridas para desempeñar las siguientes tareas administrativas:

- Autenticación, distribución de llaves, y seguridad en las comunicaciones de datos.
- Consultas de la configuración de la red de sensores y el estado de los nodos, y re-configuración de la red de sensores.
- Encendido y apagado de los nodos sensores.
- Movimiento de los nodos sensores.
- Sincronización temporal de los nodos sensores.
- Intercambio de datos relacionados a los algoritmos de determinación de posición.
- Introducción de las reglas relacionadas a la recolección de datos, nombrado basado en atributos, y agrupación en los nodos sensores.

1.5.7.2. TADAP *Task Assignment and Data Advertisement Protocol*

La principal función de TADAP es proveer al software de usuario, interfaces eficientes para la diseminación de interés, simplificando las operaciones de bajo nivel.

Este interés se puede referir a cierto atributo de un fenómeno o el inicio de cierto evento. Los usuarios envían su interés a un nodo sensor, un subconjunto de los nodos, o a toda la red.

Otra estrategia es el anunciar los datos disponibles, en el cual los nodos sensores comunican la disponibilidad de datos a los usuarios y estos a su vez realizan las peticiones de los datos en los que están interesados.

1.5.7.3. SQDDP *Sensor Query and Data Dissemination Protocol*

La misión de SQDDP es el proveer aplicaciones de usuario con interfaces para realizar peticiones, responderlas, y almacenar las respuestas entrantes. Estas peticiones no se las hacen a nodos en particular; se prefiere la selección por nombrado basado en atributos o el nombrado basado en ubicación.

Una petición basada en atributos sería: "las ubicaciones de los nodos que sensan humedad relativa arriba de 50 %". Una petición basada en ubicación: "la luminosidad detectada por los nodos de la región 1".

1.5.7.4. Protocolos Propietarios vs Standard

La Figura 1.10 muestra la comparación entre protocolos propietarios (de Texas Instruments) y standard.

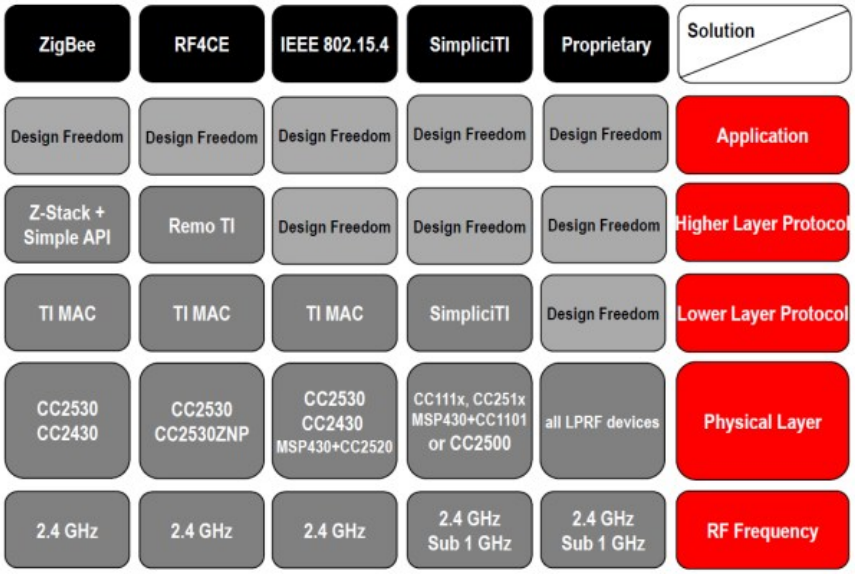


Figura 1.10: Protocolos Propietarios vs Standard.

1.6. Aplicaciones de las WSN

La WSN puede consistir en muchos tipos diferentes de sensores, como pueden ser sísmicos, magnéticos, térmicos, acústicos, radar, IR, etc. Los distintos tipos de sensores existentes pueden monitorizar una gran variedad de condiciones ambientales, que incluyen:

- Temperatura, humedad, presión.
- Condiciones de luz, movimiento de vehículos, niveles de ruido.
- Composición del suelo.
- Presencia o ausencia de cierto tipo de objetos.
- Niveles de estrés mecánico en objetos (maquinaria, estructuras, etc.).
- Características de velocidad, dirección y tamaño de un objeto.

Figura 1.11: Aplicaciones para WSN.

Los nodos sensores pueden adoptar diversas formas de trabajo, pueden actuar en modo continuo, por detección de eventos, por identificación de eventos, toma de datos localizados o como control local de actuadores (idóneo para aplicaciones domóticas).

Si tenemos el tipo de monitorización que van a realizar los sensores, podemos hacer una primera clasificación de aplicaciones, en tres tipos distintos que tendrían las siguientes propiedades:

- **Monitorización del entorno**

  Este tipo de aplicaciones se caracterizan por un gran número de nodos sincronizados que estarán midiendo y transmitiendo periódicamente en entornos puede que inaccesibles, para detectar cambios y tendencias.

  La topología es estable y no se requieren datos en tiempo real, sino para análisis futuros. Ejemplos: control de agricultura, microclimas, etc.

- **Monitorización de seguridad**

  Son aplicaciones para detectar anomalías o ataques en entornos monitorizados continuamente por sensores. No están continuamente enviando datos, consumen menos y lo que importa es el estado del nodo y la latencia de la comunicación: se debe informar en tiempo real.

  Como ejemplos tenemos el control de edificios inteligentes, detección de incendios, aplicaciones militares, seguridad, etc.

- **Tracking**

  Aplicaciones para controlar objetos que están etiquetados con nodos sensores en una región determinada. La topología aquí va a ser muy dinámica debido al continuo movimiento de los nodos: se descubrirán nuevos nodos y se formarán nuevas topologías.

- **Redes híbridas**

  Son aquellas en las que los escenarios de aplicación reúnen aspectos de las tres categorías anteriores.

El concepto de microsensores comunicados de forma inalámbrica promete muchas nuevas áreas de aplicación. De momento las vamos a clasificar en: militares, entorno, salud, hogar y otras áreas comerciales. Por supuesto es posible ampliar esta clasificación.

1.6.1. Aplicaciones militares

Las WSNs pueden ser parte integral de sistemas militares C4ISRT (command, control, communications, computing, intelligence, surveillance, reconnaissance and targeting) que son los que llevan las órdenes, el control, comunicaciones, procesamiento, inteligencia, vigilancia, reconocimientos y objetivos militares.

El rápido y denso despliegue de las redes de sensores, su autoorganización y tolerancia a fallos las hace una buena solución para aplicaciones

militares. Ofrecen una solución de bajo coste y fiable para éstas ya que la pérdida de un nodo no pone en riesgo el éxito de las operaciones.

Ejemplos de aplicación de este área son: monitorización de fuerzas aliadas, equipamiento y munición; reconocimiento del terreno y fuerzas enemigas; adquisición de blancos; valoración de daños; reconocimiento de ataques nucleares, biológicos y químicos, etc.

1.6.2. Aplicaciones medioambientales

En este campo tenemos aplicaciones como el seguimiento de aves, animales e insectos; monitorización de condiciones ambientales que afectan al ganado y las cosechas; irrigación; macroinstrumentos para la monitorización planetaria de gran escala; detección química o biológica; agricultura de precisión (monitorización de niveles de pesticidas, polución y erosión del terreno); detección de incendios; investigación meteorológica o geofísica; detección de inundaciones; mapeado de la biocomplejidad del entorno; y estudios de la polución.

1.6.3. Aplicaciones sanitarias

Proveer interfaces para los discapacitados; monitorización integral de pacientes; diagnósticos; administración de medicamentos en hospitales; monitorización de los movimientos y procesos internos de insectos u otros pequeños animales; telemonitorización de datos fisiológicos humanos; y seguimiento y monitorización de pacientes en un hospital.

1.6.4. Aplicaciones del hogar

Los nodos sensores pueden ser introducidos en aparatos domésticos como aspiradoras, microondas, hornos, heladeras y televisores. Esto permite que sean manejados remotamente por los usuarios finales mediante una comunicación que se realizaría vía satélite o Internet.

A través de las redes de sensores pueden crear hogares inteligentes donde los nodos se integran en muebles y electrodomésticos. Los nodos dentro de una habitación se comunican entre ellos y con el servidor de la habitación. Estos servidores de habitaciones se comunican también entre ellos dando así conectividad entre distintas habitaciones.

1.6.5. Otras aplicaciones comerciales

Otras aplicaciones comerciales son la monitorización de la fatiga de materiales; teclados virtuales de edificios; gestión de inventario; monitorización

de la calidad de productos; construcción de oficinas inteligentes; control ambiental en edificios de oficinas; control de robots y guiado en entornos de fabricación automática; juguetes interactivos; museos interactivos; control y automatización de procesos; monitorización de desastres; estructuras inteligentes; máquinas de diagnóstico; transporte; control local de actuadores; detección y monitorizado de robo de coches, etc.

Capítulo 2

El Protocolo SimpliciTI^{TM}

2.1. Introducción

SimpliciTI^{TM} es un protocolo de radiofrecuencia de bajo consumo propietario de Texas Instruments, pero cuyo código se encuentra liberado. Utiliza baja potencia y es capaz de integrar redes que pueden llegar a los 30 nodos. Al poder los nodos permanecer "dormidos" durante largos intervalos de tiempo hace que SimpliciTI^{TM} sea un protocolo de baja potencia, también es de bajo coste al hacer uso de memorias flash inferiores a los 4 kBytes y memorias RAM menores de 512 Bytes. Por lo tanto, se puede decir que SimpliciTI reúne las principales propiedades que se pueden esperar para desarrollar aplicaciones que requieren comunicaciones inalámbricas de corot alcance, de bajo consumo, bajo costo y relativamente de fácil puesta en marcha.

SimpliciTI^{TM} ha sido diseñado para una fácil implementación, usando los mínimos recursos requeridos por parte de los microcontroladores. Por ello funciona perfectamente en los microcontroladores y transceptores utilizados en este trabajo.

Las características principales de SimpliciTI^{TM} son:

- Bajo requerimiento de disponibilidad de memoria (8kB Flash y 1kB RAM, dependiendo de la configuración).
- Advance network control (seguridad, radio frequency agility).
- Soporta *sleeping modes*.

Para más información sobre el protocolo SimpliciTI^{TM} ver el Apéndice B.

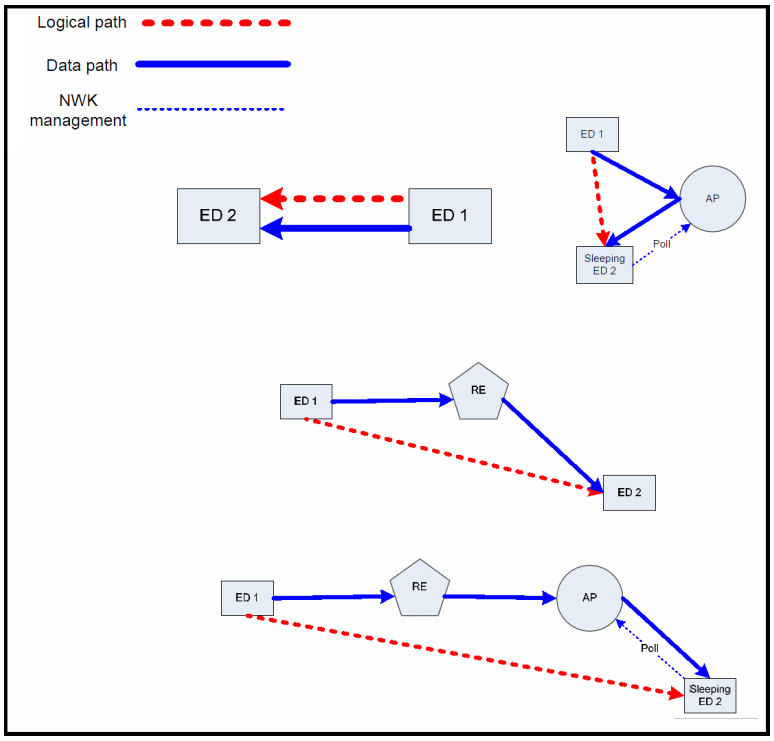


Figura 2.1: Diferentes topologías de red soportadas por SimpliciTI.

2.2. Arquitectura de red

2.2.1. Introducción

SimpliciTI™ soporta dispositivos finales, denominados ED *(End Devices)* que son los dispositivos que se comunican con el AP *(Acces Point)* encargado de identificar a todos los EDs de la red asignándoles un identificador cuando entran en comunicación con él, de este modo cuando un ED se comunique para enviar paquetes sabrá su precedencia según el identificador que anteriormente le ha otorgado. Los mensajes serán almacenados por el AP. Una opción que proporciona SimpliciTI™ es extender la red mediante extensores de rango, RE *(Range Extender)* que permite ampliar la distancia con hasta cuatro saltos.

2.2.2. Topologías de red

Soporta 2 topologías básicas: peer-to-peer y star, en donde el hub es un peer a todos los otros dispositivos de la red. La Figura 2.1 muestra las diferentes topologías soportas por el protocolo.

2.2.3. Tipos de dispositivos

SimpliciTI™ permite al usuario implementar tres tipos de dispositivos: End Device, Range Extender y Access Point.

2.2.3.1. End Device

Es el elemento más básico de la red. Generalmente contiene la mayoría de los sensores y/o actuadores de la red. Una red peer-to-peer se compone exclusivamente de end devices (y eventualmente de range extenders).

2.2.3.2. Access Point

Soporta la mayoría de las características del stack de protocolo y funciones como store-and-forward, para sleeping End Devices, administración de dispositivos de red en términos de permisos de acceso a la red, permisos para conectarse, claves de seguridad, etc. También soporta la funcionalidad propia del End Device.

En la topología en estrella, el AP actúa como el hub de la red.

2.2.3.3. Range Extender

Estos dispositivos son utilizados para repetir paquetes de modo de extender el rango de la red. Debido a la función que desempeñar, siempre están activos.

2.2.4. Capas del protocolo SimpliciTI

SimpliciTI™ esta organizado en 3 capas, como muestra la Figura 2.2:

- Data Link/Physic.
- Network.
- Application.

2.2.4.1. Application

Esta es la única capa que el desarrollador necesita implementar. Es donde es desarrolla la aplicación en si misma (por ejemplo, para manejar los sensores), e implementar la comunicación de red, mediante la utilización del protocolo SimpliciTI™, en particular de las APIs o las aplicaciones de red.

Hay que tener en cuenta, que es en esta capa en donde el desarrollador debe implementar un algoritmo de transporte confiable, ya que no existe una capa de Transporte.

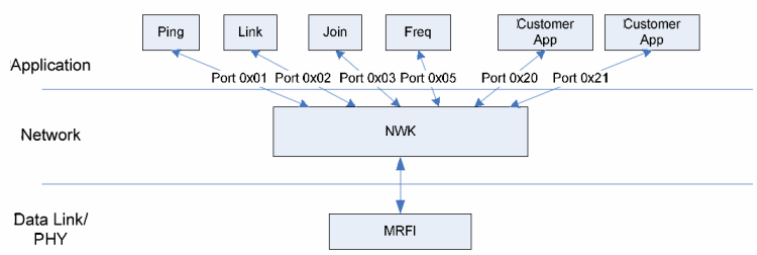


Figura 2.2: Capas del protocolo SimpliciTI.

2.2.4.2. Network

Esta capa se encarga de manejar las colas de Rx y Tx, y por otro lado, despachar los paquetes o frames a sus destinos. El destino es siempre una aplicación designada mediante un número de puerto o *Port number*.

Las aplicaciones de red o *network applications* son objetos peer-to-peer internos para manejar a la red. Estas trabajan sobre puertos predeterminados y no están pensados para ser usados por el usuario (excepto el Ping, para propósito de debugging). Su uso depende, en definitiva, del tipo de dispositivo que se trate. Estas aplicaciones son las siguientes:

- Ping (Port 0x01): para detectar la presencia de un dispositivo específico.
- Link (Port 0x02): para soportar la administración del enlace entre dos dispositivos pares.
- Join (Port 0x03): para permitir el acceso a la red en topologías con APs.
- Security (Port 0x04): para cambiar información de seguridad, como claves de encriptación y contexto de encripción.
- Freq (Port 0x05): para cambiar de canal de RF, solicitar un cambio de canal, o solicitar un echo.
- Mgmt (Port 0x06): puerto para administración general para manejar el dispositivo.

Los archivos fuentes de la capa de red se encuentran en la carpeta `Componentes/simpliciti`.

2.2.4.3. Data Link/Physic

Esta capa puede dividirse en 2 entidades:

- BSP (Board Support Package): permite abstraer la interfaz SPI de las llamadas de la capa de red que interactúan con la radio (`Componentes/bsp`).
- MRFI (Minimal RF Interface): encapsula las diferencia entre todos los hardware de las radios soportadas, hacia la capa de red (`Componentes/mrfi`).

2.3. Hardware soportado

2.3.1. Radios

SimpliciTI^{TM} soporta 5 familias de radios de Texas Instruments:

- Family 1: CC1100, CC1101, CC2500.
- Family 2: CC2510, CC2511, CC1110, CC1111.
- Family 3: CC2520.
- Family 4: CC2430.
- Family 5: CC2530.

Los archivos fuentes para estas 5 familias de radio se encuentran en la carpeta `mrfi/radios`.

2.3.2. Microcontrolador

SimpliciTI^{TM} soporta 2 familias de microcontroladores: Intel 8051 y TI MSP430 (directorio de archivos fuentes: `bsp/mcus`).

2.3.3. Placas

Las siguientes placas son soportadas: CC2430DB, CC2530EM, EXP461x, EZ430RF, RFUSB, SRF04EB, SRF05EB (directorio de archivos fuentes: `bsp/boards`).

2.3.4. Dispositivos

SimpliciTI^{TM} también soporta LEDs y/o pulsadores o switches conectados a pines GPIO del microcontrolador. Pero no hay soporte nativo para otros periféricos como la UART, LCD drivers, etc (directorio de archivos fuentes: `bsp/drivers`).

2.4. APIs

Las APIs (del inglés, Application Programming Interface) permiten al usuario implementar una red confiable con poco esfuerzo. Pero se debe tener en cuenta que tal cosa termina resultando en sacrificar flexibilidad por simplicidad.

A continuación se detallan las diferentes APIs soportas por SimpliciTI^{TM}:

- Initialization

  - smplStatus_t SMPL_Init(uint8 (*pCB)(linkID))

- Linking (bidirectional by default)

  - smplStatus_t SMPL_Link(linkID_t *linkID)
  smplStatus_t SMPL_LinkListen(linkID_t *linkID)

- Peer to peer messaging

  - smplStatus_t SMPL_Send(linkID_t lid, uint8 *msg, uint8 len)
  - smplStatus_t SMPL_Receive(linkID_t lid, uint8 *msg, uint8 len)

- Configuration

  - smplStatus_t SMPL_Ioctl(ioctlObject_t object, ioctlAction_t action, void *val)

Capítulo 3

Prueba de Concepto

3.1. Introducción

La prueba de concepto que se describe en este capítulo, tiene el propósito de demostrar una aplicación real de una "Red de Sensores Inalámbricos Autoalimentados", y en particular de una *red de sensores inalámbricos de temperatura.* La aplicación hace uso del protocolo de comunicación inalámbrica de código abierto SimpliciTI™, de Texas Instruments, tratado en el Capítulo 2 para crear una red relativamente simple en la que los *end devices, EDs*[1] transmiten la información de temperatura, voltaje y estado obtenida mediante los sensores correspondientes a un nodo central de la red llamado *network access point, AP*[2]. El Access Point transmite toda la información recolectada a través de una UART disponible a un puerto COM de la PC. Este puerto es utilizado luego por una aplicación MS Windows® que ha sido desarrollada en lenguaje C#, que mediante una interface de usuario gráfica (GUI) permite visualizar toda la información de la red en tiempo real.

Desde el punto de vista del hardware, cada nodo inalámbrico ha sido construido sobre la plataforma de Texas Instruments™ eZ430-RF2500, tanto para los *end devices* como para el *network access point*, salvando la diferencia de que éste último nodo tiene un puerto USB que permite la interacción con la PC.

[1] Es el elemento más básico de la red. Son unidades autónomas que generalmente constan de un microcontrolador, una fuente de energía, sensores/actuadores y un radiotransceptor.

[2] Este dispositivo soporta múltiples funciones como guardar y retransmitir (del inglés store-and-forward) para dispositivos "durmiente" (del inglés, sleeping end devices), administración de la red en términos de manejo de permisos de acceso, claves de seguridad, etc. El Access Point (AP) soporta también la funcionalidad de *end device*. Para el caso de una topología en estrella, actúa como el hub de la red.

Por último, se utilizará un Packet Sniffer[9], para capturar los paquetes que son enviados en la WSN.

3.2. La WSN propuesta

La prueba de concepto basada en una red de sensores inalámbricos autoalimentados de temperatura fue creada sobre la base de una arquitectura de hardware ya existente de Texas Instruments llamada eZ430-RF2500, conformada básicamente por un microcontrolador MSP430, una transceptor de RF de bajo consumo, en particular el CC2500, y un protocolo inalámbrico de bajo consumo y de código abierto, SimpliciTI®. Más adelante se procederá a detallar la plataforma eZ430-RF2500 con mayor profundidad.

El proyecto de demostración se divide en cinco partes claramente diferenciadas:

1. La alimentación de los nodos ED y AP.
2. La plataforma de hardware existente eZ430RF2500.
3. El protocolo SimpliciTI^{TM}, de Texas Instruments.
4. La aplicación del sistema embebido (EDs y AP).
5. La aplicación MS Windows® para la presentación gráfica y animada de los datos.

En vistas de tener una visión más clara de la red, a continuación se ofrecen una serie de imágenes que muestran a cada componente de la misma.

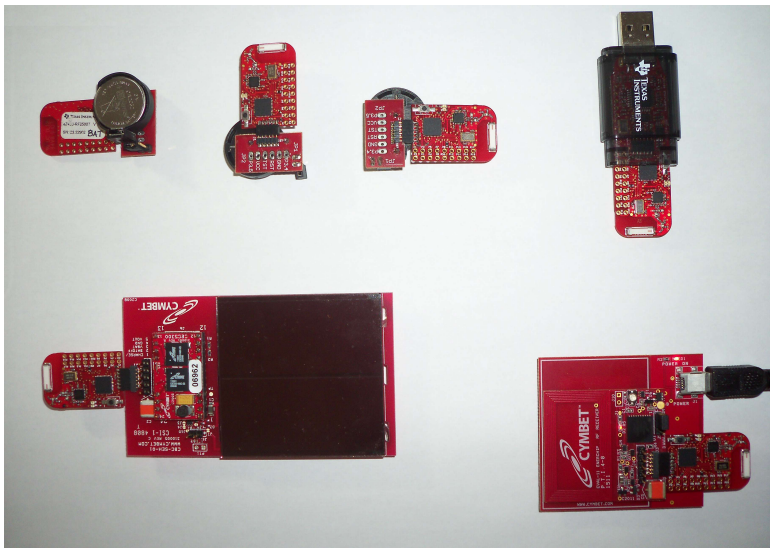


Figura 3.1: Imagen general de todos los componentes de la red.

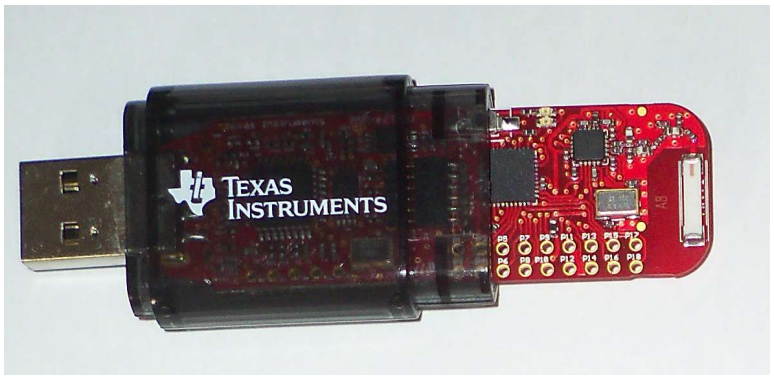


Figura 3.2: Imagen del Access Point.

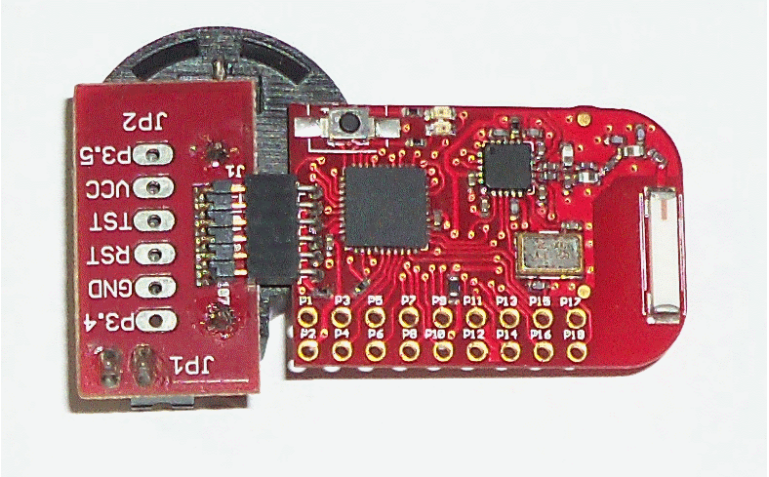


Figura 3.3: Imagen de un nodo alimentado a baterías CR2032.

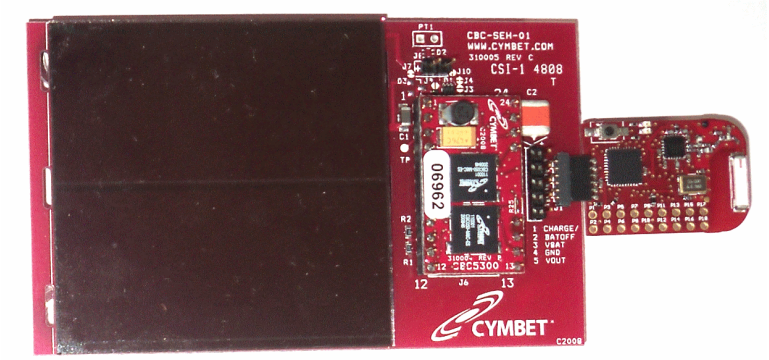


Figura 3.4: Imagen de un nodo alimentado a energía solar.

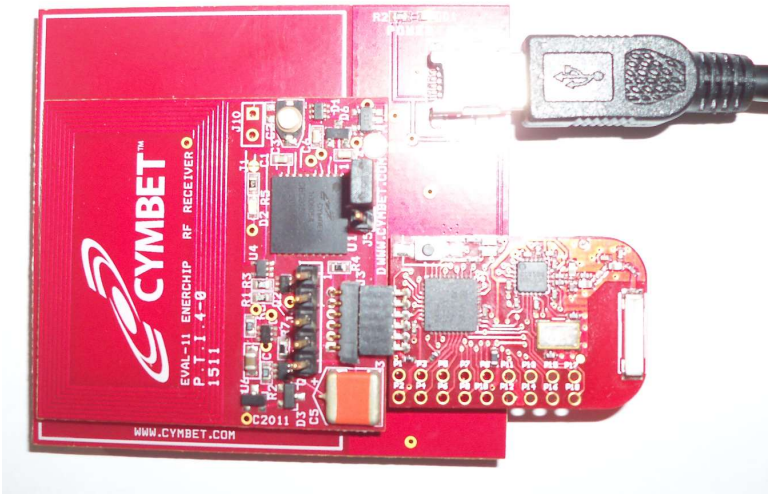


Figura 3.5: Imagen de un nodo alimentado por medio de RF.

3.2.1. Componentes de la WSN

La Figura 3.6 muestra la arquitectura a nivel de bloques de un mote autoalimentado. Los bloques componentes principales se detallan a continuación:

1. **Energy Harvester**: permite capturar energía del ambiente, ya sea esta en forma de luz, calor, movimiento, RF, etc. La salida de este bloque genera un voltaje apropiado para el próximo bloque, que administra la energía obtenida.

2. **Energy Storage & Power Management**: se encarga básicamente de administrar y almacenar la energía obtenida por el Harvester. Aquí pueden incluirse etapas de conversión de voltage, cargadores de baterías y/o capacitores, etc. Además, el bloque genera los niveles de tensión que requiere el mote para operar.

3. **Sensor(s)**: este bloque representa a todos los sensores que dispone el mote para medir variable del entorno como la temperatura, la posición, el estado, etc. Los sensores se conectan generalmente al MCU para su procesamiento.

4. **Ultra-Low Power MCU**: es la unidad de procesamiento del mote. Debe ser estrictamente de muy bajo consumo para garantizar la factibilidad técnica en términos energéticos de la correcta operación del mote. Contiene el código asociado al protocolo de comunicación inalámbrica que se utilice y la capa de aplicación. Pueden correr sistemas operativos de tiempo real, RTOS, cuando sea requerido.

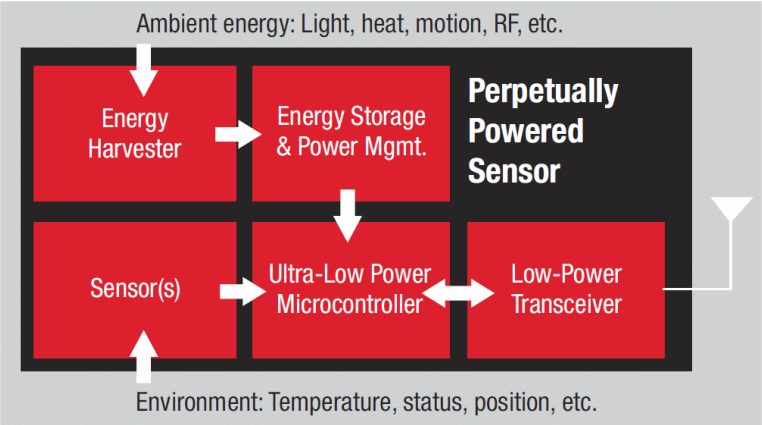


Figura 3.6: Diagrama de bloques de un mote autoalimentado.

5. **Low-Power Transceiver**: representa el radio-chip, y más aún, toda la etapa de RF del mote. Hay diferentes versiones con diferentes capacidades. Las antenas pueden ser de montaje superficial, antenas dibujadas en el mismo PCB o también pueden ser externas. Los transceivers poseen modos de operación de bajo consumo, lo que permite fundamentalmente garantizar las comunicaciones inalámbricas de bajo consumo en motes autoalimentados.

3.3. El TI eZ430-RF2500 y sus componentes

En la siguientes subsecciones se hará una breve introducción al microcontrolador de Texas Instruments MSP430 y luego, se procederá a detallar las características principales de la plataforma de hardware ez430RF2500.

3.3.1. Introducción al MSP430f2274

El corazón de esta plataforma es su microcontrolador MSP430, de Texas Instruments. Hay una familia completa de microcontroladores MSP430, las diferentes variantes se relacionan con las diferentes capacidades en cantidad de RAM/ROM y de E/S, principalmente. El microcontrolador usado en la plataforma ez430-RF2500 es el MSP420f2274, con 32 KB de memoria Flash (ROM) y 1 KB de RAM.

El MSP430 es un microcontrolador de 16-bit de arquitectura RISC. De 16 bits significa que todos los registros tienen 16 bits, la interconexión entre los elementos del micro-controlador se realiza mediante buses de 16 bits. RISC (Set Reducido de Instrucciones) se refiere al hecho de que hay sólo 27 instrucciones básicas.

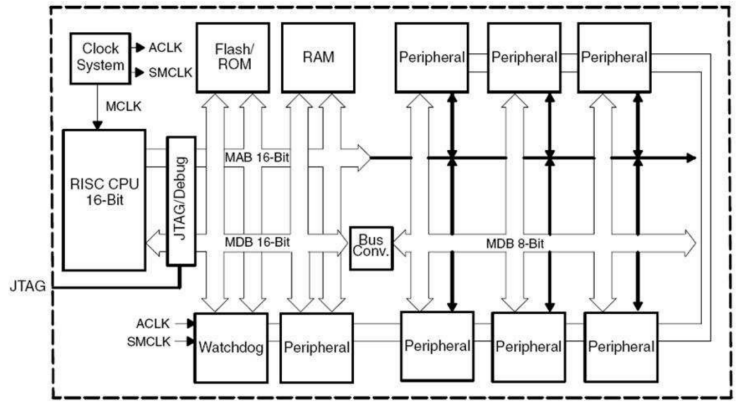


Figura 3.7: Estructura interna del MSP430.

3.3.2. Operación del MSP430

La Figura 3.7 muestra la arquitectura interna del MSP430. La CPU contiene 16 registros; su funcionamiento se detalla a continuación. Un sistema de reloj genera pulsos a una velocidad programable (por ejemplo, 1MHz), por lo que cada μs una instrucción se recupera de la memoria (ROM), se copia en el registro correspondiente y se ejecuta[3]. Un ejemplo de ejecución puede ser la adición de dos registros y copiar el resultado en un tercero. En la práctica, estos detalles de bajo nivel son atendidos por el compilador, que traduce lenguaje C en lenguaje ensamblador y código binario.

3.3.3. Programando el MSP430

Cuando se programa un *mote*[4], lo que se programa es su microcontrolador, es decir, se termina colocando el código binario compilado en el lugar correcto en la memoria ROM del MSP430. Cuando se enciende la alimentación del mote, el MSP430 empieza por ir a buscar la primera instrucción en un lugar predeterminado en la memoria ROM; aquí es en realidad, donde la herramienta programación (entorno de programación) pone el código compilado.

Las herramientas típicas para programar MSP430, son el IAR Workbench, y el TI CodeComposer.

Ahora, para configurar otros componentes (por ejemplo, para definir la potencia de transmisión de la radio), es necesario programar al MSP430

[3]En sentido estricto, las instrucciones pueden tomar más de un ciclo de CPU para ejecutarse

[4]Es un nodo en una red de sensores inalámbricos, que es capaz de realizar algún procesamiento, obtener cierta información de sensores, y comunicarse con otros nodos en la red.

de tal manera que configure la radio al principio del programa, es decir, del código embebido. Este es un concepto importante, ya que se aplica a la mayoría de los motes que tienen en su arquitectura de hardware, dos chips principales, uno es un MCU y otro es un transceiver (radiochip), y se encuentran comunicados en casi todos los casos por una interfaz serial. Vale mencionar, que existen arquitecturas basadas en SoCs (del inglés, System on Chip), que tienen ambos componentes en una misma pastilla de silicio.

3.3.4. Interrupciones

Un programa para un sensor inalámbrico es, en la práctica, una serie de piezas muy pequeñas de código que se ejecutan cuando ocurre algún evento: por ejemplo, cuando se pulsa el botón, se procede a encender el LED rojo. Cuando ocurre un evento provocado por un cambio de estado, por ejemplo, un pulsador conectado a uno de los puertos del MSP430 (puerto P1.2 en el caso del botón del módulo eZ430-RF2500), será necesario programar el MSP430 de tal manera que el cambio de la situación en el puerto P1.2 genere una interrupción. Cuando se genera una interrupción, el MSP430 interrumpe su ejecución actual (si existe), y comienza a ejecutar una función específica llamada la rutina de servicio de la interrupción (ISR, del inglés, Interrupt Service Routine) asociada a la interrupción en particular. Una vez que esta función ha finalizado (normalmente un ISR es una función muy pequeña), continúa con su ejecución normal (si existe, y decimos si existe ya que el MCU puede encontrarse simplemente en estado de bajo consumo).

Para la aplicación de demostración que se desarrolla en el presente trabajo, es importante tener en cuenta en concepto de programación orientada a interrupciones. Ellos nos permitirá entender las aplicaciones que son programadas bajo un perfil de bajo consumo energético, es decir, utilizando un perfil de bajo ciclo de trabajo (LDCP, del inglés, Low-Duty Cycle Profile).

3.3.5. Timers

Al escribir el código, será necesario esperar un tiempo antes de realizar alguna tarea en particular (por ejemplo, cuando se recibe un paquete, esperar 10 ms, y enviar un paquete de respuesta). Esto se puede hacer ,haciendo uso de un temporizador o timer, un componente específico del MSP430. Físicamente, un timer es un registro de 16 bits que se incrementa en cada ciclo de reloj[5], es decir, una vez cada μs con un reloj de 1MHz. La cuenta comienza en 0, y cuenta con hasta un valor programable, a partir del cual se genera una interrupción del timer, se resetea a 0, y empieza a contar de nuevo.

[5]En rigor, los timers se pueden configurar para contar de forma ascendente, descendente, ascendente/descendente, ver [12]

3.3.6. Funcionalidades de I/O

El MSP430 utilizado en la plataforma de hardware del mote tiene 40 pines:

- 4 tienen funciones analógicas y para alimentación del mote;
- 2 son utilizados para testing en la fábrica;
- 2 son usados si un cristal externo es utilizado como fuente de clock, lo cual no es el caso de la plataforma eZ430-RF2500;
- 32 pines tienen funciones digitales.

Los 32 pines digitales se encuentran agrupados en 4 puertos de 8 pines cada uno. Cada pin tiene un nombre de la forma P$x.y$, y representa la posición del pin dentro del puerto x. Todos los pines pueden ser entradas y salidas genéricas, y una serie de registros de 8-bit son usados para configurarlos adecuadamente:

- PxDIR.y define la dirección del puerto P$x.y$; salida si P$x.y$=1, entrada si P$x.y$=0;
- PxOUT.y setea el estado del puerto P$x.y$ cuando se encuentra configurado como salida;
- PxIN.y lee el estado del puerto P$x.y$ cuando se encuentra seteado como entrada;
- PxIE.y habilita las interrupciones en dicho puerto;

Cada unos de estos registros almacena 8 bits, uno para cada pin. Como resultado, P1DIR=0b11110000[6] significa que los pines del P1.1 al P1.4 son entradas, mientras de P1.5 a P1.8 son salidas. Para setearresetear un pin determinado, es necesario usar los operadores binarios que se presentan en la Figura 3.8.

Debe notarse que la mayoría de los 32 pines digitales también pueden ser usados para funciones específicas (interfase SPI, entrada al conversor AD, ...), ver [11] para más detalles.

[6] 0bx significa que x esta escrito en binario; 0xx significa que x esta escrito en hexadecimal. Por lo tanto tenemos 0x1A=0b00011010.

A	=	0b01101001		
∼A	=	0b10010110		
A	—=	0b00000010	⇒	A=0b01101011
A	&=	∼0b00001000	⇒	A=0b01100001
A	∧=	0b10001000	⇒	A=0b11100001
A	<<	2	⇒	A=0b10100100
A	>>	2	⇒	A=0b00011010

Figura 3.8: Operadores binarios usados para setearresetear bits individuales.

3.3.7. Operación de Bajo Consumo

Mientras que el MSP430 queda a la espera por las interrupciones, es importante reducir su consumo de energía durante los períodos de inactividad apagando los clocks que no se están utilizando. Cuantos más clocks se apaguen, menos energía se utiliza, pero obviamente hay que asegurarse de mantener activos aquellos que son estrictamente necesarios. En el MSP430 hay cuatro modos de baja consumo (LPM1, . . . , LPM4) los cuales apagan determinadas fuentes de clock. (ver detalles en [11]).

En la práctica, sólo es necesario dejar encendido el clock auxiliar que da señal se clock a un timer para "despertar" al MSP430 después de algún tiempo predeterminado. Esto se logra al poner al MSP430 en modo de bajo consumo 3 (LPM3), añadiendo la siguiente línea de código al final de la función `main`:

```
__bis_SR_register(LPM3_bits);
```

Todos los periféricos internos del MSP430 han sido cuidadosamente diseñados para ser inteligente y con características que reducen la carga de la CPU. Por ejemplo, los conversores AD internos ofrecen una la exploración automática de canales y un controlador de acceso directo a memoria o DMA, para cargar datos directamente en la memoria sin necesidad de utilizar el CPU. Esto permite un mayor rendimiento del sistema y reduce la energía consumida.

La transparencia de la arquitectura del MSP430 se puede experimentar rápidamente con su revolucionario CPU ortogonal de 16-bit que tiene un procesamiento más eficaz, es de menor tamaño y más eficiente en código que otros microcontroladores de 8/16-bit.

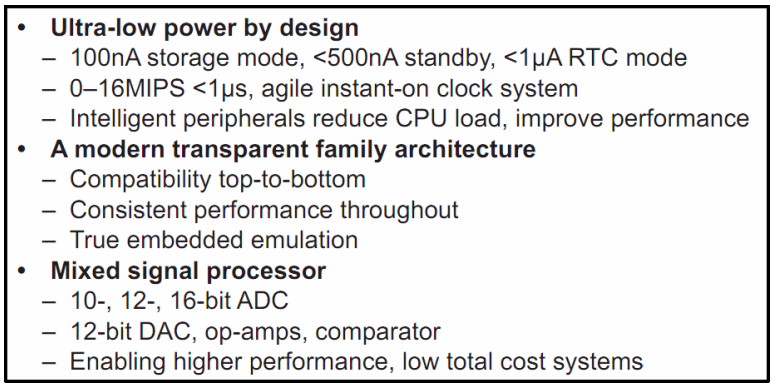


Figura 3.9: Características destacadas del MSP430 (en inglés).

En la Figura 3.11 se presentan las características más importantes del MSP430 en términos de consumo energético para diferentes modos de operación.

La familia MSP430 está orientada a conseguir un bajo consumo. Es por ello que resulta idónea para el control de transceptores y procesadores ZigBee/802.15.4. Siempre que el microprocesador está esperando una interrupción es de vital importancia apagar los relojes que no estemos usando si queremos disminuir el consumo. Para ello se definen cuatro modos de bajo consumo (LPM, del inglés, Low Power Modes) que se diferencian en el número de relojes de que prescinde cada uno. Podemos ver dichos modos de bajo consumo en la Figura 3.10.

El consumo de corriente del microprocesador es proporcional tanto a la frecuencia de trabajo como a la tensión de alimentación. Así, para los valores representados en la tabla, se ha supuesto una tensión de alimentación de 3V y una frecuencia de trabajo de 8MHz en el modo activo, que son los valores con los que trabajaremos en nuestro proyecto. Para unos valores de consumo más detallados se recomienda acudir a la hoja de datos [12]. MCLK (Main Clock) es el reloj principal usado por la CPU, SMCLK (Submain Clock) es un reloj secundario y ACLK (Auxiliary Clock) es un reloj auxiliar. Podemos ver su distribución en la arquitectura interna anteriormente mostrada.

En la práctica sólo dejando activo el reloj auxiliar ACLK, que despierta el integrado MSP430 después de un determinado intervalo de tiempo, es suficiente. Esto se consigue usando el modo LPM3.

El MSP430 ha sido diseñado específicamente para aplicaciones de medición alimentadas mediante baterías, con todo lo que ello implica. En el modo LMP3, el MSP430 consume típicamente corrientes en el orden de μA. Pero no sólo es importante que el MCU ofrezca prestaciones de bajo consumo para que una aplicación también lo sea. Es muy importante diseñar la aplicación del sistema embebido utilizando lo que se denomina "Ultra-Low

Modo	Consumo	Características
Activo	2,8 mA	CPU activada Todos los relojes activos
LPM0	90 µA	CPU desactivada ACLK y SMCLK activos MCLK inactivo
LPM1	90µA	LPM0 + generador de continua de DCO inactivo si DCO no se usa durante el modo activo
LPM2	25µA	CPU desactivada MCLK y SMCLK inactivos generador de continua de DCO activo ACLK activo
LPM3	1µA	CPU desactivada MCLK y SMCLK inactivos generador de continua de DCO inactivo ACLK activo
LPM4	0,1µA	CPU desactivada MCLK, SMCLK y ACLK inactivos generador de continua de DCO inactivo Oscilador de cristal inactivo

Figura 3.10: Modos de bajo consumo en el microprocesador MSP430.

Power Activity Profile" (ULPAP), o perfil de ultra bajo consumo de energía, mediante la utilización de un "Low Duty Cycle Profile" (LDCP), o perfil de bajo ciclo de trabajo. Estos son básicamente el mismo concepto y es esencial a la hora de desarrollar aplicaciones que deben ser alimentadas a baterías con una vida útil de funcionamiento esperada en el rango de años, o bien para aplicaciones autoalimentadas mediante técnicas de "energy harvesting".

La Figura 3.12 muestra los consumos asociados a una implementación LDCP. Como se puede observar, el *duty cycle* es extremadamente bajo, donde el tiempo en el que el sistema permanece en modo activo, es decir, con mayor consumo energético, es muy pequeño en comparación con el tiempo en stand-by, en donde el consumo energético es notablemente bajo (en el caso del ejemplo, unas 250 veces menor).

Así es como se ha desarrollado e implementado la aplicación de demostración de la red WSN presentada en este trabajo. En los momentos en donde el sistema permanece en modo activo (modo que es activado mediante un timer que genera una interrupción por fin de cuenta), se realizan las transmisiones de RF de los valores que han sido sensados, mediante una comunicación digital. Tanto el MCU como la radio, permanecen luego en modo bajo consumo hasta que una nueva interrupción haya sido generada.

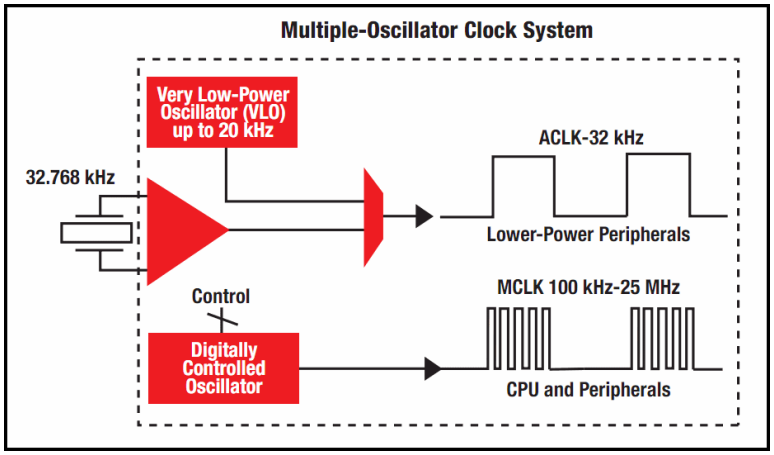


Figura 3.11: Diagrama simplificado del sistema múltiple de osciladores del MSP430.

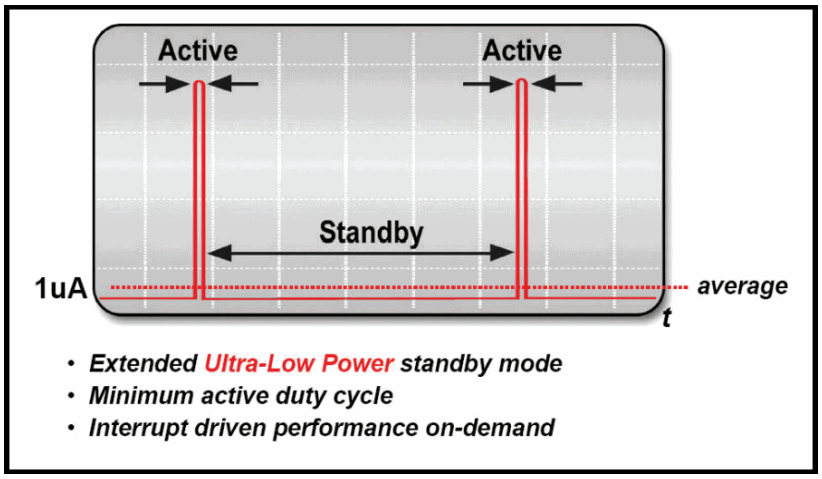


Figura 3.12: Ejemplo de un perfil de bajo consumo.

3.3.8. El ez430-RF2500

En primer lugar detallaremos todo el material que viene con la plataforma eZ430-RF2500 que aparte de un CD con distintos programas como el IAR y ejemplos de códigos para el microcontrolador MSP430 encontramos todos los elementos de hardware que podemos ver en la Figura 3.13 y 3.14.

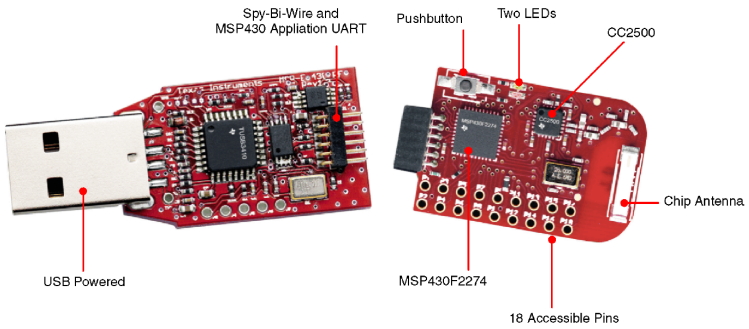


Figura 3.13: ez430-RF2500: programador/emulador y target board.

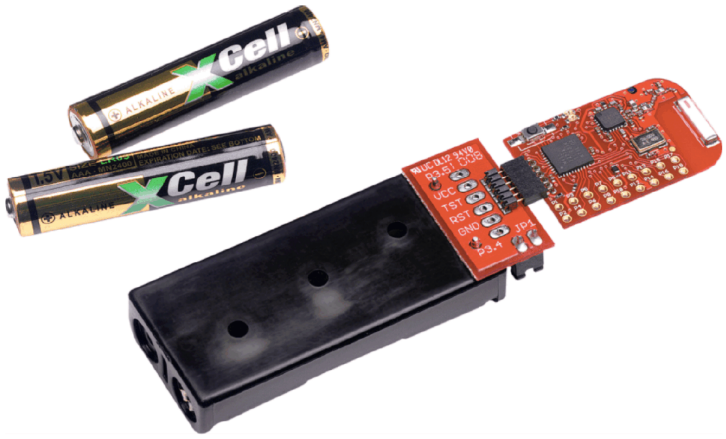


Figura 3.14: ez430-RF2500: target board.

Esta plataforma es una completa herramienta de desarrollo de software para proyectos inalámbricos (wireless). Utiliza una interface USB lo que facilita la interacción a la hora de manipular el microcontrolador de ultra-bajo consumo MSP430F2274 a 16Mhz y el transceptor inalámbrico CC2500 a 2.4Ghz.

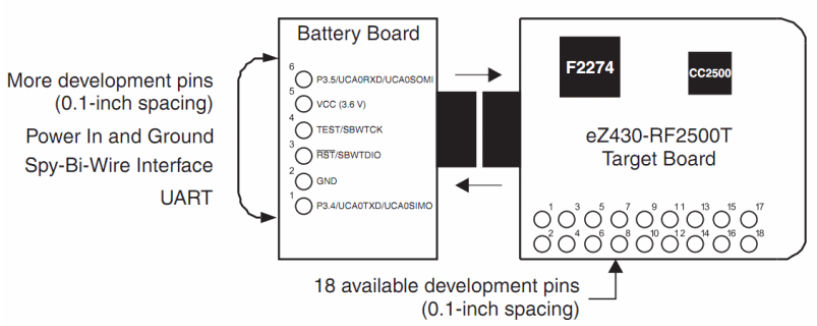


Figura 3.15: ez430-RF2500: esquema de pines accesibles.

La "target board" tiene accesibles los pines más utilizados de una manera más accesible que teniendo que soldarlo directamente en la placa como se puede ver en la Figura 3.15 y 3.17 donde se describe con detalle la función de cada uno de los pines disponibles.

Además se muestran a en la Figura 3.16 continuación las descripciones de los pines en ambas placas.

La placa eZ430-RF2500T que se muestra en la Figura 3.18 incluye los siguientes componentes:

- Microcontrolador MSP430F2274: 32kB de memoria Flash, 1kB de RAM, USCI (UART, 2xSPI, I2C, IrDA), 10-bit 200ksps ADC, 2 amplificadores operacionales.
- Transceiver CC2500: 2.4GHz, banda multicanal ISM de baja potencia.
- Dos LEDs, uno rojo y uno verde, ambos de montaje superficial.
- Un pulsador.

Cada placa o *target board* puede ir conectada indistintamente o al soporte de las baterías (2 pilas AAA) o a la interface USB para su alimentación o a cualquier módulo de energy harvesting. La interface USB permite recibir y enviar de forma remota información a la PC utilizando la aplicación UART del MSP430.

Pin	Function	Description
1	GND	Ground reference
2	VCC	Supply voltage
3	P2.0 / ACLK / A0 / OA0I0	General-purpose digital I/O pin / ACLK output / ADC10, analog input A0
4	P2.1 / TAINCLK / SMCLK / A1 / A0O	General-purpose digital I/O pin / ADC10, analog input A1 Timer_A, clock signal at INCLK, SMCLK signal output
5	P2.2 / TA0 / A2 / OA0I1	General-purpose digital I/O pin / ADC10, analog input A2 Timer_A, capture: CCI0B input/BSL receive, compare: OUT0 output
6	P2.3 / TA1 / A3 / VREF– / VeREF– / OA1I1 / OA1O	General-purpose digital I/O pin / Timer_A, capture: CCI1B input, compare: OUT1 output / ADC10, analog input A3 / negative reference voltage output/input
7	P2.4 / TA2 / A4 / VREF+ / VeREF+ / OA1I0	General-purpose digital I/O pin / Timer_A, compare: OUT2 output / ADC10, analog input A4 / positive reference voltage output/input
8	P4.3 / TB0 / A12 / OA0O	General-purpose digital I/O pin / ADC10 analog input A12 / Timer_B, capture: CCI0B input, compare: OUT0 output
9	P4.4 / TB1 / A13 / OA1O	General-purpose digital I/O pin / ADC10 analog input A13 / Timer_B, capture: CCI1B input, compare: OUT1 output
10	P4.5 / TB2 / A14 / OA0I3	General-purpose digital I/O pin / ADC10 analog input A14 / Timer_B, compare: OUT2 output
11	P4.6 / TBOUTH / A15 / OA1I3	General-purpose digital I/O pin / ADC10 analog input A15 / Timer_B, switch all TB0 to TB3 outputs to high impedance
12	GND	Ground reference
13	P2.6 / XIN (GDO0)	General-purpose digital I/O pin / Input terminal of crystal oscillator
14	P2.7 / XOUT (GDO2)	General-purpose digital I/O pin / Output terminal of crystal oscillator
15	P3.2 / UCB0SOMI / UCB0SCL	General-purpose digital I/O pin USCI_B0 slave out/master in when in SPI mode, SCL I2C clock in I2C mode
16	P3.3 / UCB0CLK / UCA0STE	General-purpose digital I/O pin USCI_B0 clock input/output / USCI_A0 slave transmit enable
17	P3.0 / UCB0STE / UCA0CLK / A5	General-purpose digital I/O pin / USCI_B0 slave transmit enable / USCI_A0 clock input/output / ADC10, analog input A5
18	P3.1 / UCB0SIMO / UCB0SDA	General-purpose digital I/O pin / USCI_B0 slave in/master out in SPI mode, SDA I2C data in I2C mode

Figura 3.16: ez430-RF2500: pinout de la target board.

Pin	Function	Description
1	P3.4 / UCA0TXD / UCA0SIMO	General-purpose digital I/O pin / USCI_A0 transmit data output in UART mode (UART communication from 2274 to PC), slave in/master out in SPI mode
2	GND	Ground reference
3	$\overline{\text{RST}}$ / SBWTDIO	Reset or nonmaskable interrupt input Spy-Bi-Wire test data input/output during programming and test
4	TEST / SBWTCK	Selects test mode for JTAG pins on Port1. The device protection fuse is connected to TEST. Spy-Bi-Wire test clock input during programming and test
5	VCC (3.6V)	Supply voltage
6	P3.5 / UCA0RXD / UCA0SOMI	General-purpose digital I/O pin / USCI_A0 receive data input in UART mode (UART communication from 2274 to PC), slave out/master in when in SPI mode

Figura 3.17: ez430-RF2500: pinout de la battery board.

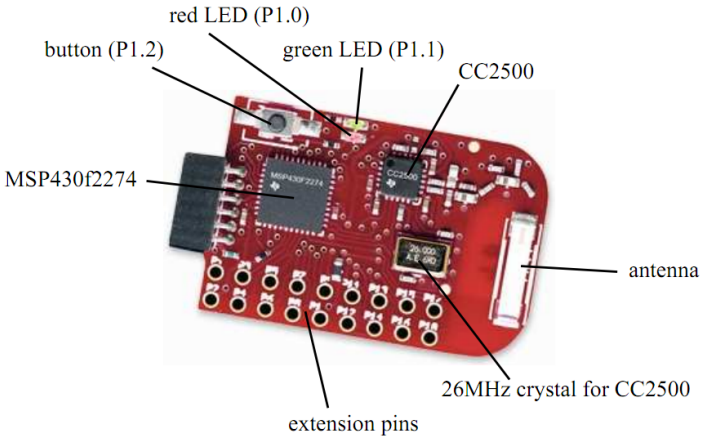


Figura 3.18: ez430-RF2500: descripción de la target board.

3.3.8.1. El CC2500

El CC2500 es un transceptor diseñado para aplicaciones de bajo consumo a 2.4Ghz. Su circuitería trabaja dentro de la banda de frecuencia ISM para usos industriales, científicos y médicos (entre 2400MHz a 2483.5MHz).

Soporta varios tipos de modulación llegando hasta 500Kbaudios. Es capaz de almacenar datos, trabajar con paquetes e indicar la calidad del canal de enlace (el RSSI, del inglés, Received Signal Strength Indicator).

Los parámetros principales de las operaciones realizadas en el CC2500 pueden ser controlados con una interface SPI. La gestión de cola de datos en el buffer del transceptor es gestionada mediante la política FIFO. Suele ser usado junto con un microcontrolador y otros componentes pasivos, como justamente es el caso de la plataforma ez430-RF2500, el que se encuentra conectados mediante la interfaz SPI el MSP430 y el CC2500, como se muestra en la Figura 3.19.

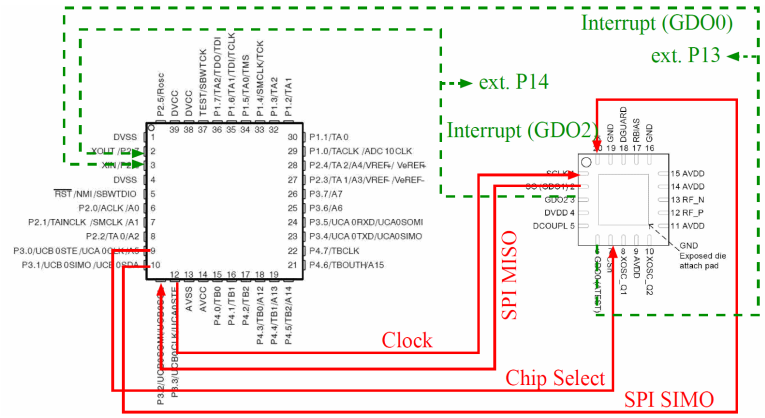


Figura 3.19: Interconexión entre el MSP430 y el CC2500 en el ez430-RF2500.

El transceptor CC2500 tiene diferentes estados en los cuales realiza funciones diferentes:

- Transmitir.
- Recibir.
- Idle: preparado para recibir, pero no lo hace (algunas funciones del hardware se desactivan y consume menos energía).
- Sleep: todas las partes del transceptor están apagadas. Hay que tener en cuenta que en ocasiones despertar el sistema puede consumir más energía que dejarlo en Idle.

Las características principales del transceptor CC2500 son:

- Alta sensibilidad (-104 dBm a 2,4 baudios, 1 % de tasa de error).
- Bajo consumo de corriente (13,3 mA en RX, 250 baudios, de entrada muy por encima de la sensibilidad límite).
- Potencia de salida programable hasta +1 dBm.
- Tasa de datos programable desde 1,2 a 500 kBaudios.
- Rango de frecuencia: 2400 a 2483,5 MHz.
- Técnicas de modulación soportadas: OOK, 2-FSK, GFSK, y MSK.
- 90 μs settling time, del sintetizador de frecuencia.
- Sensor de temperatura integrado.

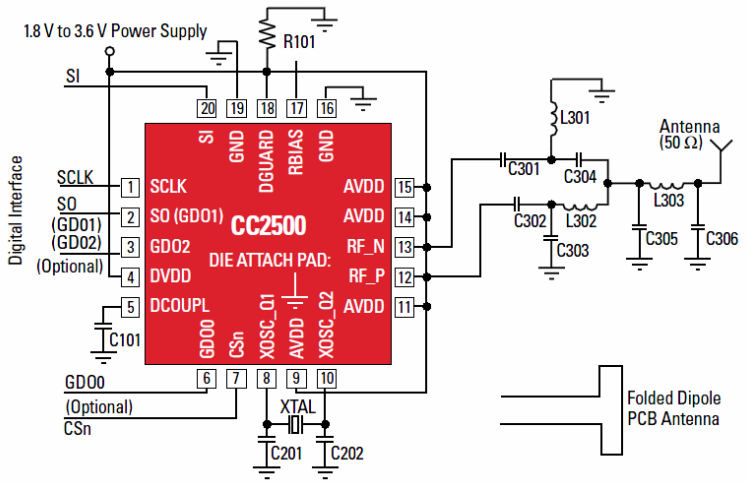


Figura 3.20: Pinout del CC2500.

- Soporta AFC (Automatic Frequency Compensation), RSSI digital, CSI (Carrier Sense Indicator), CCA (Clear Channel Assessment) antes de la transmisión.

Respecto de las características de operación del CC2500, la Figura 3.21 presenta los valores más sobresalientes.

Parameter (433/868 MHz, 3.0 V, 25°C)	Min	Typ	Max	Unit	Condition
Operating conditions					
Frequency range	2400	—	2483.5	MHz	
Data rate (programmable)	1.2	—	500	kBaud	
Sensitivity, 2.4 kBaud	—	–104	—	dBm	Optimized sensitivity
Sensitivity, 250 kBaud	—	–89	—	dBm	Optimized sensitivity
Output power (programmable)	–30	—	1	dBm	
Operating supply voltage	1.8	—	3.6	V	
Current consumption					
RX input signal at the sensitivity limit, 2.4 kBaud	—	17.0	—	mA	
RX input well above sensitivity limit, 2.4 kBaud	—	14.5	—	mA	
RX input signal at the sensitivity limit, 250 kBaud	—	16.6	—	mA	Current optimized
RX input well above sensitivity limit, 250 kBaud	—	13.3	—	mA	Current optimized
Current consumption, TX, 0 dBm	—	21.2	—	mA	
Current consumption, TX, –12 dBm	—	11.1	—	mA	
Current consumption, power down	—	<1	—	µA	

Figura 3.21: Condiciones de operación del CC2500.

3.4. La aplicación de monitoreo, WSN Console

La aplicación para monitorear y mostrar gráficamente a la red de sensores inalámbricos ha sido desarrollada enteramente en el lenguaje C#, utilizando el entorno de desarrollo MS Visual Studio C# Express 2010. Para mayor información acerca de la implementación, referirse al Apéndice D.

Figura 3.22: Pantalla de inicio de la consola.

La consola permite mostrar el estado de la red casi en tiempo real (como ya se ha explicado anteriormente, para lograr una buena eficiencia energética se hace uso del LDCP, por lo que el sistema no tiene una respuesta en tiempo real, sino que tiene una cierta latencia asociada al ciclo de trabajo definido en la red), mostrando en el centro de la pantalla siempre al Access Point, dibujado en color rojo. La información se presenta mediante círculos que parpadean al ritmo de actualización de la red, y todos muestran la información medida de temperatura ambiente y voltaje de alimentación.

Teniendo en cuenta que es una red de sensores inalámbricos autoalimentados, se ha propuesto generar una identificación de colores para distinguir las diferentes fuente de energía de cada End Device.

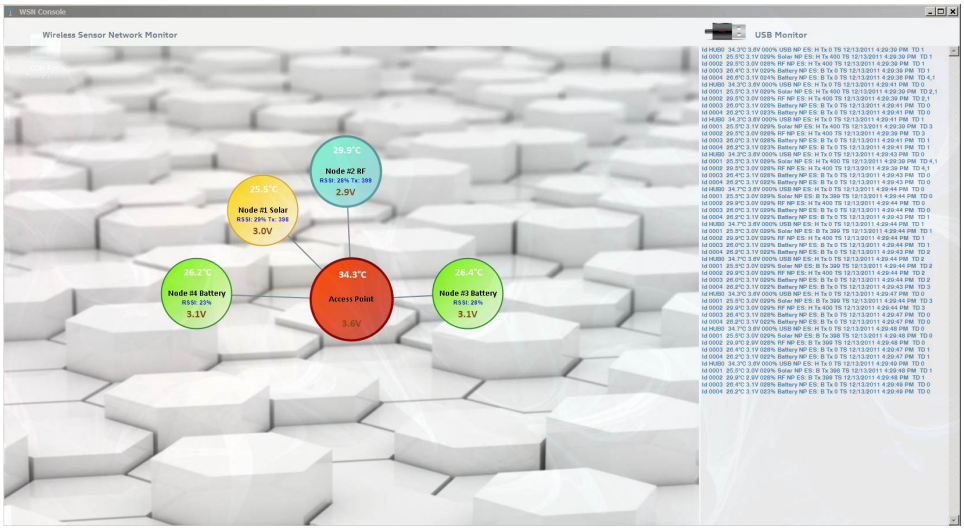


Figura 3.23: Pantalla principal mostrando varios nodos en la red.

Las referencias de colores se detallan a continuación:

- Rojo: identifica una fuente de energía externa, por ejemplo, desde el puerto USB.
- Verde: identifica que la fuente de energía proviene de una batería.
- Celeste: identifica que la fuente de energía proviene de ondas EM.
- Amarillo: identifica que la fuente de energía proviene de la energía solar.

Además del código de colores, existe una parámetro adicional: cuando un nodo es autoalimentado por medio medio de técnicas de *energy harvesting*, como es el caso del "nodo solar" o del "nodo de RF", éstos tienen al mismo tiempo una batería o elemento de almacenamiento de energía que permite almacenar la energía recolectada del medio, para su uso posterior, cuando la fuente de energía externa, no se encuentre presente (i.e. falta de luz solar). Por ello surge la necesidad de poder identificar si un nodo que dispone de un *harvester* se encuentra operando desde su batería o desde el harvester. Así, cuando estos nodos toman la energía directamente del harvester, se mostrará un ***cuadrado en lugar de un círculo***, y viceversa cuando tomen su energía de la batería interna.

Figura 3.24: Pantalla principal mostrando un ED con su harvester activo.

Adicionalmente, los nodos que poseen un harvester, mostrarán información acerca de la cantidad de transmisiones que pueden realizar con la energía que han recuperado del medio (vale aclarar que el cálculo es estimativo). Por último, se mostrará también el nivel de voltaje de la batería interna.

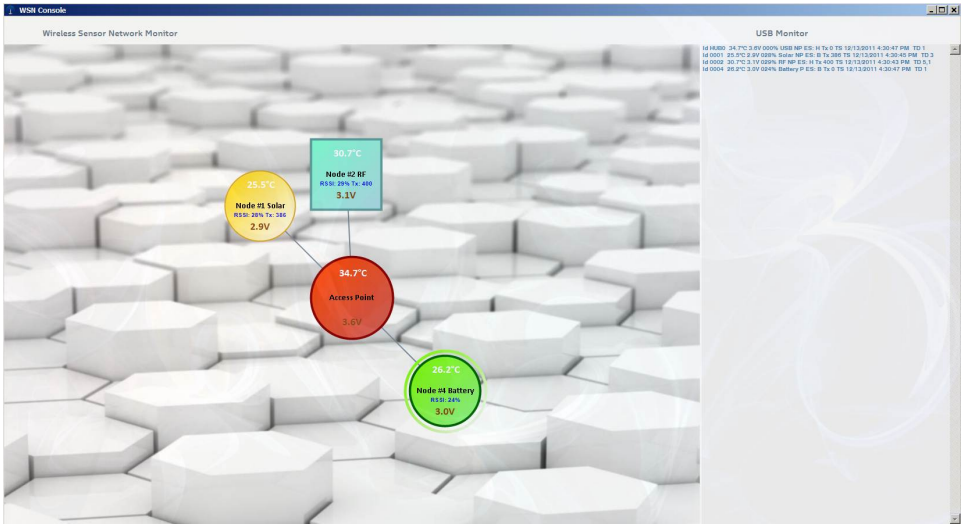


Figura 3.25: Pantalla principal mostrando la identificación de un nodo.

Otras de las características de la consola, es que dado que el transceiver TI CC2500 posee una indicador digital de RSSI (Indicador del Nivel de Señal Recibido), ello nos permite hacer una estimación de cuán cerca o cuán

lejos se encuentra un ED del AP. Por ello, los EDs se "moverán" hacia el AP o en sentido contrario de acuerdo al valor de RSSI que cada uno mida, información que también será representada en pantalla.

Todos los paquetes recibidos por el AP serán directamente parseados por la consola y mostrados en un panel a la derecha de la pantalla.

Para demostrar la interacción entre un ED y la consola, se ha implementado en todas las capas del sistema, la funcionalidad de poder identificar físicamente a un nodo en la pantalla. Esto se logra mediante el pulsador que tiene cada ED, de modo que al pulsarlo, aparecerá un círculo concéntrico al nodo del cual se ha presionado su pulsador, para identificarlo en pantalla. Al presionar nuevamente, este desaparecerá.

3.5. El firmware del sistema embebido

La red se encuentra básicamente compuesta por dos tipos de dispositivos, el End Device y el Access Point. Cada uno de ellos esta programado con un firmware distinto, ya que los End Devices son nodos que se conectan y/o desconectan en cualquier momento, y se encargan de transmitir la información al nodo central de la red, que es el Access Point, el cual recibe la información de cada uno de los End Devices y la re-transmite por un puerto serie a la PC.

Desde el punto de vista del protocolo SimpliciTI, un Access Point(AP) maneja la red y desde el punto de vista energético, siempre esta activo, recibiendo paquetes de uno o mas SimpliciTI End Devices (ED), una vez por segundo. Los EDs de las WSN, contienen sensores que permiten obtener las magnitudes medidas por la red, y pasan la mayor parte del tiempo en *low power mode 3* (LPM3), despertándose una vez por segundo para tomar muestras de los canales A/D, que se traducen en muestras de temperatura y voltaje, y luego envían dicha información al AP de la red por RF.

Para mayor información acerca de la implementación, ver el código fuente tanto del ED como del AP, en el Apéndice E.

3.5.1. El Access Point (AP)

La primer tarea que ejecuta el firmware del AP, es la inicialización del protocolo SimpliciTI, y la de definir a este nodo justamente como el nodo central de la red o Access Point. Luego, haciendo uso del sensor de temperatura interno y el ADC10, el AP comienza a medir la temperatura ambiente y el voltaje de alimentación del USB una vez por segundo y transmite ésta información a la PC. Además, el AP se mantiene "escuchando" continuamente por otros EDs que quieran unirse a la red y por paquetes de EDs que ya han sido agregados a la misma. Utilizando los LEDs indicadores de la plataforma ez430-RF2500, el AP notifica de las dos transacciones que ocurren en la red, principalmente:

- LED Rojo: indica la transmisión a la PC de las mediciones del AP.
- LED Verde: indica que se ha recibido una paquete desde algunos de los EDs en presentes en la red.

3.5.2. El End Device (ED)

Cuando se alimenta al nodo, el ED comienza inmediatamente a "buscar" un AP al cual conectarse. Mientras realiza esta búsqueda, tanto el LED rojo como el verde parpadean. Cuando el AP ha sido "descubierto" el ED intenta unirse a la red, mientras que el LED rojo parpadea para indicar este intento

de conexión. Si no puede conectarse con el AP, continuará parpadeando el LED rojo. Una vez que se haya podido conectar al AP, todos los LEDs se apagarán, y el ED entrará en modo LPM3, efectuando con una corto destello del LED verde cuando realiza una transmisión y por ende, se encuentre en modo activo.

Capítulo 4

Conclusiones

En esta tesis se ha desarrollado una demostración de la factibilidad técnica de la implementación de una red de sensores inalámbricos autoalimentados mediante la utilización de técnicas de energy harvesting (haciendo uso de energías como la solar y la electromagnética) y haciendo foco en una aplicación real que sirvió de caso de estudio: la medición de temperatura ambiente, en diferentes puntos del espacio tridimensional.

Se ha tomado como plataforma de hardware base al TI ez460-RF2500 y como protocolo de red inalámbrica de bajo consumo, al TI SimpliciTI. Por sobre estas capas, se ha implementado la aplicación propuesta, programando tanto la solución del sistema embebido como la aplicación MS Windows^{TM} para la representación gráfica del estado de la red WSN en "tiempo real".

Desde el punto de vista técnico, se ha comprobado que es totalmente factible disponer de nodos inalámbricos autoalimentados ya sea por energía solar o mediante ondas de RF, y mediante la aplicación de perfiles de bajo consumo, evitar el uso de baterías y extender la vida útil de los nodos, en el orden de años (obviamente dependiendo de los requerimientos de actualización de la red).

También se ha verificado, la flexibilidad que otorga el uso de radios digitales de bajo consumo y que económicamente, es viable su implementación en aplicaciones de gran escala.

Desde el punto de vista académico-práctico, el presente trabajo deja abierta una puerta enorme hacia la introducción de la redes de sensores inalámbricos y las técnicas de energy harvesting en los materiales de cátedra, tema que no forma parte de la currícula. Si bien este trabajo se ha enfocado en puntos muy específicos, permitirá tomarlo como base para desarrollar

una innumerable cantidad de aplicaciones.

En el ámbito del I+R&D, el stack de protocolos de las redes de sensores inalámbricas es un campo de investigación aún muy abierto. Se necesita hacer mucha investigación en cada uno de los componentes de la arquitectura (capas y planos) para mejorar el desempeño de este tipo de redes venciendo las restricciones que limitan el crecimiento de las redes de sensores.

Finalmente, quiero decir que este trabajo pone fin a una larga carrera, con éxitos y fracasos, y a una etapa maravillosa de la vida.

Bibliografía

[1] "Wireless Sensor Networks, Signal Processing and Communications Perspectives", Edited by Ananthram Swami Army Research Laboratory, USA Qing Zhao University of California at Davis, USA Yao-Win Hong National Tsing Hua University, Taiwan, John Wiley & Sons Ltd, 2007.

[2] "Synchronization in a wireless sensor network designed for surveillance applications", José A. Sánchez fernández, Ana B. García Hernando, José F. Martínez Ortega, Lourdes López

[3] "Self-Correcting Time Synchronization Using Reference Broadcast In Wireless Sensor Network, Fengyuan Ren And Chuang Lin, Tsinghua University Feng Liu, Beihang University, IEEE Wireless Communications, August 2008"

[4] "A distributed consensus protocol for clock synchronization in wireless sensor network, Luca Schenato, Giovanni Gamba, Proceedings of the 46th IEEE Conference on Decision and Control New Orleans, LA, USA, Dec. 12-14, 2007"

[5] "Fine-Grained Network Time Synchronization using Reference Broadcasts, Jeremy Elson, Lewis Girod and Deborah Estrin Department of Computer Science, University of California, Los Angeles, 2002"

[6] "Implementación de un prototipo de red de sensores inalámbricos para invernaderos, Carlos Alberto Castillo Luzón, Escuela Politécnica Nacional, Quito, 2007"

[7] "Timing-sync Protocol for Sensor Networks Saurabh Ganeriwal Ram Kumar Mani B. Srivastava Networked and Embedded Systems Lab (NESL), University of California Los Angeles, 56-125B Eng. IV, UCLA EE Dept., Los Angeles, CA 90095, 2003"

[8] "Wireless Sensor Monitor Using the eZ430-RF2500, Application Report, Miguel Morales, Zachery Shivers, Texas Instruments, April, 2011"

[9] "Texas Insturments SmartRF Protocol Packet Sniffer"

[10] "`http://www.webopedia.com/TERM/E/embedded_system.html`"

[11] "MSP430x2xx Family User's Guide, 2008, SLAU144E"

[12] "MSP430x22x2, MSP430x22x4 Mixed Signal Microcontroller, 13 May 2009, SLAS504C"

[13] "CC2500, CC2500, Low-Cost Low-Power 2.4 GHz RF Transceiver (Rev. B), Texas Instruments, Inc., 19 May 2009, data Sheet SWRS040C"

[14] "eZ430-RF2500 Development Tool User's Guide, Texas Instruments, April 2009, SLAU227E"

[15] "Programación de Redes de Sensores Inalámbricos para aplicaciones domóticas, Pedro José Meseguer Copado, Universidad Politécnica de Cartagena, 2007"

Apéndice A

Introducción a IEEE 802.15.4 (por Gonzalo Campos Garrido)

Capítulo 2

ZigBee/802.15.4

2.1. Introducción a ZigBee/802.15.4

Como ya hemos visto en el capítulo 1, ZigBee/802.15.4 es una tecnología inalámbrica de bajo consumo, baja tasa de transmisión y coste reducido. Se engloba dentro del grupo de las tecnologías WPAN (*Wireless Personal Area Network*). El estándar ha sido definido por la *ZigBee Alliance*, de la que forman parte más de 300 empresas, y por el IEEE *Task Group* 4. En concreto, la *ZigBee Alliance* se encarga de la definición de las capas de red y aplicación mediante el estándar ZigBee [9] mientras que el *Task Group* 4 se centra en la definición de las capas física y MAC (*Medium Access Control*) mediante estándar 802.15.4 [10]. Estos términos, ZigBee y 802.15.4, son a veces utilizados como sinónimos cuando en realidad definen partes diferentes de la pila de protocolos. En la figura 2.1 podemos ver una simplificación de dicha pila de protocolos.

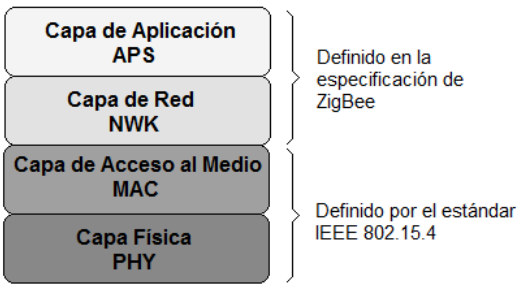


Figura 2.1: Pila de protocolos ZigBee/802.15.4

Esta tecnología es idónea para formar redes de sensores. Los posibles usos de ZigBee/802.15.4, como principal exponente de las redes inalámbricas de bajo consumo, han sido expuestos en la sección 1.2.

Dentro del estándar ZigBee encontramos dos versiones: ZigBee-2006 y ZigBee PRO (2007). En las siguientes secciones utilizaremos el término ZigBee para referirnos a ZigBee PRO. La mayoría de los fabricantes comercializan dispositivos compatibles con ambas versiones.

2.2. Topología

Antes de describir los distintos tipos de redes ZigBee es necesario definir los 3 tipos distintos de dispositivos que la componen. Estos son:

- **Coordinador.** Este dispositivo inicializa y controla la red. También se encarga de gestionar las tareas de seguridad. Para poder formar una red ZigBee debe existir un coordinador.
- ***Router*.** Estos dispositivos son los encargados de extender la cobertura de la red, gestionando nuevos caminos en el caso de que la red experimente congestión o se produzca la caída de algún nodo. Pueden conectarse directamente al coordinador o a otros *routers*. Gestionan dispositivos hijo.
- **Dispositivo final.** Este tipo de dispositivo puede enviar y recibir datos pero no realizar tareas de encaminamiento. Deben estar conectados a un *router* o al coordinador y no admiten dispositivos hijo.

Los dispositivos también pueden organizarse según otro criterio:

- **FFD *(Full Function Device).*** Este dispositivo tiene una funcionalidad completa. Puede funcionar como Coordinador, *router* o dispositivo final.
- **RFD *(Reduced Function Device).*** Tiene una implementación mínima del protocolo 802.15.4. Está pensado para realizar tareas extremadamente simples en las que se envíen pequeñas cantidades de datos. Ejemplos de esto pueden ser interruptores de luz, sensores de infrarrojos u otros dispositivos con una funcionalidad básica. Sólo pueden estar conectados a un FFD. Únicamente pueden funcionar como dispositivos finales.

Como podemos ver en la especificación de ZigBee [9], el estándar soporta 3 tipos de topologías de red: estrella, malla y árbol (figura 2.2). Una red se caracteriza por su identificador PAN (PAN ID), que la distingue de otras posibles redes funcionando en la misma zona.

2.2.1. Topología en estrella

Este tipo de redes se componen por un FFD funcionando como coordinador y varios FFD o RFD funcionando como dispositivos finales. Todos los dispositivos finales están directamente conectados al coordinador, que es el encargado de haber iniciado la red y de gestionarla. En una red tipo estrella todas las comunicaciones entre dos dispositivos finales deben pasar antes por el coordinador. Se recomienda que mientras los dispositivos finales estén alimentados por baterías el coordinador lo esté directamente a través de la red eléctrica ya que su consumo es mucho mayor. Un problema de esta configuración es que la expansión de la red está muy limitada puesto que el rango de alcance del coordinador es el que define el tamaño de la red.

2.2.2. Topología en malla

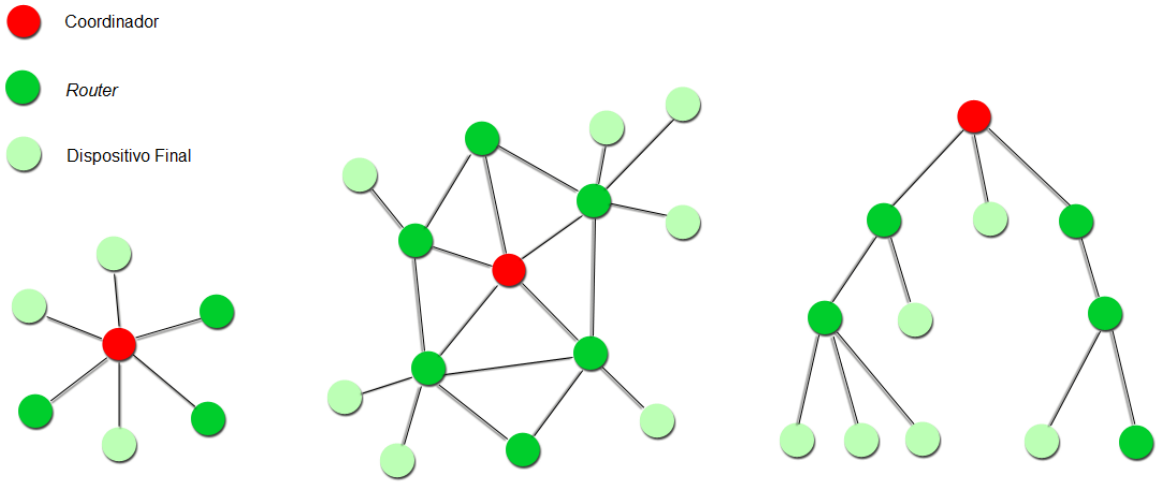
En este tipo de red se puede establecer comunicación directa entre cualquier par de nodos. El coordinador no realiza funciones muy diferentes a las que realizan el resto de *routers* de la red; de hecho, la

función de coordinador la realiza el primer *router* que forme parte de la red. Al no depender de un único dispositivo para gestionar la red, la fiabilidad de esta configuración es mayor.

En la topología en malla ganamos en flexibilidad a costa de aumentar la complejidad. Esto se debe a que para comunicar cualquier par de dispositivos hay más de un camino posible. La elección de dicho camino conlleva un aumento de computación que debe realizarse a nivel de red. Por lo tanto, estas consideraciones no se tienen en cuenta en la especificación del IEEE 802.15.4 si no que se definen en la especificación de ZigBee.

2.2.3. Topología en árbol

La topología en árbol es un caso particular de la topología en malla. En ella los diferentes componentes de la red se organizan en una estructura jerárquica. El coordinador o los *routers*, que son los encargados de gestionar el encaminamiento, pueden tener dispositivos hijos. Estos dispositivos hijos pueden ser otros *routers* o dispositivos finales. La topología en árbol, al igual que la mallada, es idónea para expandir la red de forma dinámica. Al igual que en el caso anterior se recomienda que el coordinador y los *routers* estén alimentados de forma cableada.



(a) Estrella (b) Malla (c) Árbol

Figura 2.2: Topologías de red

2.3. Capa Física

La capa física gestiona la transmisión y recepción de datos usando determinados canales radio. En 802.15.4 tenemos tres posibles bandas de frecuencia en las que poder trabajar: 868 MHz, 915 MHz y 2,4 GHz. En la primera banda únicamente tenemos un canal entre 868 y 868,6 MHz. En la siguiente banda tenemos 10 canales entre 902 y 928 MHz. Por último, 16 canales más están localizados entre 2,405 y 2,48 GHz. El protocolo permite una selección dinámica del canal, de forma que se elija el menos ruidoso de entre aquellos posibles. Lógicamente, la tasa de transferencia es distinta para cada una de las bandas de frecuencia. Para 868 MHz la tasa de transferencia es de 20 Kbps, para 915 MHz es de 40 kbps y para 2,4

GHz es de 250 kbps. Por supuesto, las bandas de 868 y 915 MHz tienen un rango de alcance mayor que la de 2,4 GHz a cambio de una menor tasa de transferencia. Otra diferencia es que, como vimos en la sección 1.1, la banda de 868 MHz está disponible sólo en Europa mientras que la de 915 MHz sólo en Estados Unidos y Australia. Sin embargo la banda a 2,4 GHz es universal, por lo que su uso está permitido en la mayoría de países del mundo. En la figura 2.3 podemos ver una representación de los canales radio 802.15.4.

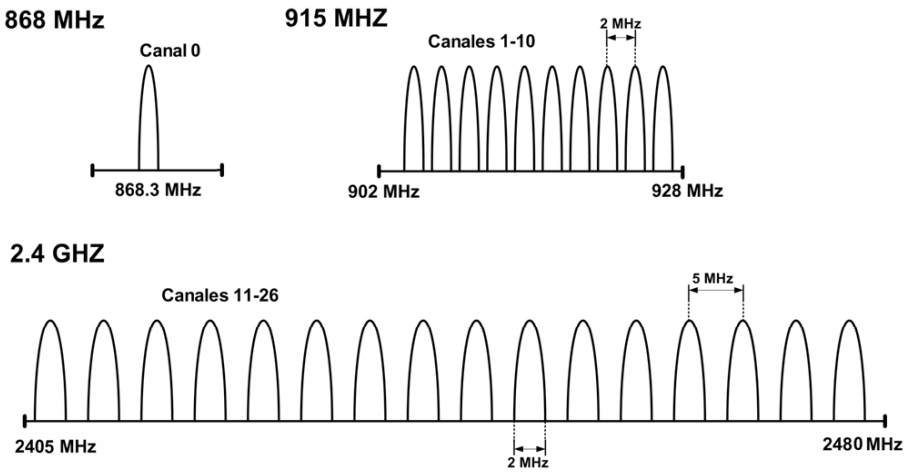


Figura 2.3: Canales radio 802.15.4. Fuente: [11]

En 804.15.4 se utiliza DSSS (*Direct Sequence Spread Spectrum*) como técnica de ensanchado de espectro. En la tabla 2.1 mostramos la tasa de transferencia y el tipo de modulación usada para las distintas frecuencias de transmisión.

Banda de frecuencia	Tasa de transferencia	Modulación
868 MHz	20 kbps	BPSK
915 MHz	40 kbps	BPSK
2,4 GHz	250 kbps	O-QPSK

Cuadro 2.1: Características de las frecuencias 802.15.4

Para configurar la capa física se emplean una serie de constantes y atributos. Podemos ver un listado exhaustivo a partir de la página 45 de [10]. Ejemplos de dichas constantes y atributos son *aMaxPHYPacketSize, aTurnaroundTime, phyCurrentChannel, phyCCAMode, etc.*

Respecto a las funciones que realiza la capa física en 802.15.4 podemos destacar:

- **Activar y desactivar el transceptor radio.** El transceptor radio tiene tres modos de funcionamiento: transmisión, recepción y *sleeping* (descanso). Bajo petición de la capa MAC la capa física debe conmutar entre estos tres estados. El estándar exige que la conmutación entre transmisión y recepción, o viceversa, se haga en menos de 12 símbolos ($aTurnaroundTime = 12$). Para el caso de 2,4 GHz y teniendo en cuenta que cada símbolo corresponde a 4 bits, se puede calcular el tiempo máximo de conmutación entre los estados de transmisión y recepción:

$$12\ símbolos = 48\ bits \tag{2.1}$$

$$\frac{48\ bits}{250\frac{kbits}{s}} = 192\ \mu s \tag{2.2}$$

- **Detección de energía de cada canal (ED, *Energy Detection*)**. La capa física también es la encargada de comprobar qué nivel de potencia tiene un determinado canal. Esta medida es utilizada por la capa de red como parte de su algoritmo para la elección de canal.
- **Indicación de la calidad de enlace (LQI, *Link Quality Indicator*)**. Mide la potencia en dBm de la señal que ha transmitido el último paquete.
- **Evaluación de canal libre (CCA, *Clear Channel Assessment*)**. Este indicador será utilizado por el algoritmo CSMA-CA (*Carrier Sense Multiple Access with Collision Avoidance*) como se verá más adelante.
- **Selección de frecuencia.** Dentro de los 27 canales posibles de ZigBee, la capa física debe ser capaz de seleccionar aquel que le especifiquen las capas superiores.
- **Envío y recepción de datos.**

De acuerdo con el estándar 802.15.4, el sistema radio debe ser capaz de emitir con una potencia de -3 dBm o superior y debe tener una sensibilidad mínima de -20 dBm.

En la figura 2.4 se muestra el formato de trama 802.15.4 a nivel físico. Dicha trama está compuesta principalmente por una cabecera de sincronización (SHR), un campo para indicar la longitud de la trama (PHR) y la carga útil.

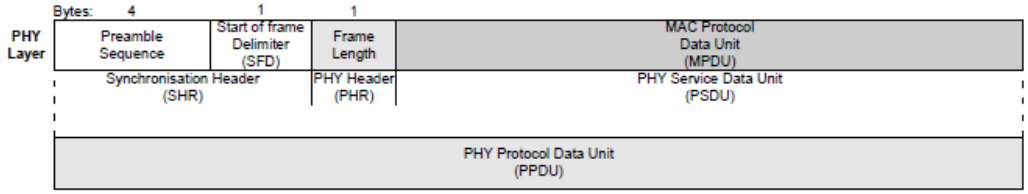


Figura 2.4: PDU (*Protocol Data Unit*) de nivel físico o PPDU. Fuente: [10]

Una vez generada la PPDU, y de acuerdo con el estándar 802.15.4 [10], los bits se convierten en símbolos. Puesto que cada símbolo está formado por 4 bits, se tienen 16 posibles símbolos. Para cada uno de ellos existe una secuencia de chips (formada por 32 bits cada una) que será la que definitivamente se module y envíe gracias al sistema de radio. El proceso se esquematiza en la figura 2.5.

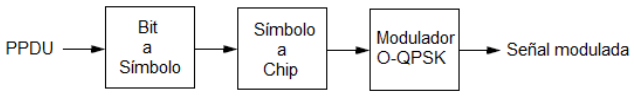


Figura 2.5: Tratamiento de la PPDU

2.4. Capa MAC

2.4.1. Descripción general

La capa MAC, o capa de acceso al medio, es la responsable de asegurar la comunicación entre un nodo y todos los nodos conectados directamente a él, evitando colisiones y mejorando la eficiencia. Más concretamente, las tareas que la capa MAC tiene que realizar son:

- Generar balizas (*beacons*) si el dispositivo es un coordinador y se funciona en modo balizado.
- Sincronizar las balizas de la red.
- Gestionar la conexión y desconexión a la red de los dispositivos asociados al propio nodo.
- Emplear el algoritmo CSMA-CA para gestionar el acceso al canal.
- Asegurar un enlace fiable con la capa MAC de los nodos contiguos.

Al igual que ocurre con la capa física, para configurar la capa MAC se utilizan una serie de constantes y atributos. Podemos ver un listado exhaustivo en la página 159 de [10]. Ejemplos de dichas constantes y atributos son *macBeaconOrder, macSuperframeOrder* o *aBaseSuperframeDuration,* que, junto con otros, serán utilizados en las siguientes secciones para profundizar en la descripción de la capa de acceso al medio.

2.4.2. Modos de funcionamiento

El protocolo MAC soporta dos modos de funcionamiento (el coordinador es el encargado de seleccionar uno u otro en el momento de iniciar la red):

- **Modo balizado**. La baliza es generada periódicamente por el coordinador y distribuida por toda la red gracias a los *routers*. Dicha baliza sirve para sincronizar todos los nodos de la red, de modo que estos puedan despertarse en un momento determinado (conocido por todos), enviar los datos almacenados y volver al modo de ahorro energético (*sleep*). Así, tanto el coordinador, como los *routers* y los dispositivos finales pueden pasar gran parte del tiempo en modo de bajo consumo. La topología en malla no admite el modo balizado debido a la complejidad que ello conllevaría (a un mismo dispositivo podrían llegarle balizas provenientes de distintos *routers*).

- **Modo no balizado**. En este modo los dispositivos no están sincronizados unos con otros. Así, únicamente los dispositivos finales pueden entrar en el modo *sleep* mientras que los *routers* y el coordinador deben estar continuamente con el sistema radio en modo recepción y así estar preparados para recibir datos en cualquier momento. Este modo es más simple pero hace que gran parte de sus nodos (el coordinador y los *routers*) tengan un mayor consumo energético. Asimismo, impide que los coordinadores puedan planificar sus envíos a los dispositivos finales.

Modo balizado

El coordinador es el que determina si trabajamos o no en este modo. En caso positivo, él será el encargado de distribuir la baliza a todos los elementos de la red.

Cuando se trabaja en el modo balizado se utiliza una estructura de envío de datos conocida como "supertrama". Una supertrama está delimitada por dos balizas consecutivas y se compone por una parte activa y otra inactiva (figura 2.6). Durante la parte activa el coordinador interactúa con la red mientras que en la inactiva entra en modo *sleep*. La estructura de una supertrama viene definida por dos parámetros:

- BO (*macBeaconOrder*). Determina el intervalo de transmisión de las balizas. El valor del BO y del BI (*Beacon Interval*) están relacionados según la siguiente expresión, donde $0 \leq BO \leq 14$ y $aBaseSuperframeDuration$ representa el número de símbolos que forman una supertrama de orden 0 (por defecto toma el valor de 960):

$$BI = aBaseSuperframeDuration \cdot 2^{BO} \; símbolos$$

- SO (*macSuperframeOrder*). Determina el tamaño de la parte activa de la supertrama. El valor del SO y del SD (*Superframe Duration*) están relacionados según la siguiente expresión, donde $0 \leq SO \leq BO \leq 14$:

$$SD = aBaseSuperframeDuration \cdot 2^{SO} \; símbolos$$

Si los valores de BI y SD coinciden la supertrama carecerá de periodo inactivo. Por otra parte, de acuerdo con el estándar 802.15.4, si SO y BO valen 15 el coordinador configurará la red para trabajar en modo no balizado.

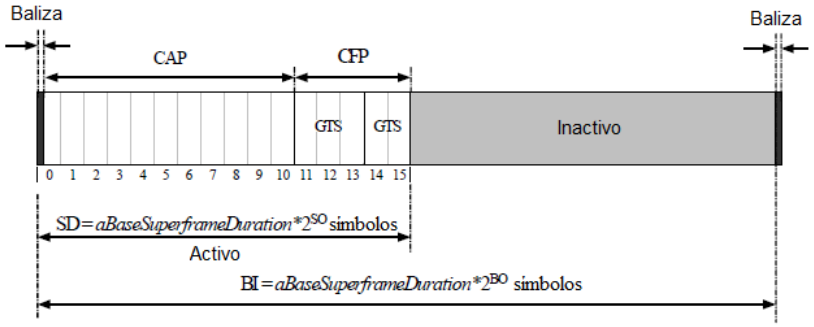


Figura 2.6: Estructura de supertrama. Fuente: [10]

La parte activa de la supertrama se divide en 16 *slots* temporales, el primero de los cuales contiene la baliza. Estos *slots* se dividen en dos grupos: el *Contention Access Period* (CAP) y el *Contention Free Period* (CFP). A continuación los detallamos:

- **CAP**. Un dispositivo que desee comunicarse con su nodo padre debe competir por el canal utilizando el algoritmo CSMA-CA. Si consigue transmitir lo hará utilizando uno de los *slots* del CAP.

- **CFP**. Bajo petición de un nodo, el coordinador puede asignarle uno o varios *slots* temporales del CFP de forma que no tenga que competir con otros nodos de la red por el envío de información. Un

grupo de *slots* temporales asignados a un determinado nodo se denominan *Guarantee Time Slots* (GTS). Mediante esta característica, el protocolo 802.15.4 puede ofrecer cierta calidad de servicio a ciertas aplicaciones con requerimientos especiales.

Modo no balizado

Como hemos dicho en el punto anterior, si se asigna el valor de 15 a las variables BO y SO activaremos el modo no balizado. En él no existen los conceptos de *slots* temporales, balizas ni supertramas. Cada nodo disputa el acceso al canal mediante el algoritmo CSMA-CA y, si lo consigue, los demás dispositivos tendrán que esperar su turno. Únicamente los mensajes de reconocimiento de mensaje recibido (*acknowledgment* o *ack* en inglés) podrán ser enviados inmediatamente y sin necesidad de competir por acceder al medio mediante CSMA-CA.

En este modo el consumo del coordinador y los *routers* es mucho mayor que en el modo balizado. El consumo de los dispositivos finales sí se mantiene prácticamente igual en ambos modos puesto que siempre pueden pasar al modo de bajo consumo una vez transmitidos los datos. Viéndolo de otra forma podemos decir que en una red con balizas todos los dispositivos están dormidos la mayor parte del tiempo, gracias a la sincronización entre ellos que les permite despertarse a la vez. En una red sin balizas el coordinador y los *routers* no tienen manera de saber cuándo se despertarán los dispositivos finales, por lo que tienen que estar el 100 % del tiempo con el sistema radio activado.

El modo no balizado nos ofrece simplicidad a cambio de un consumo mayor y una menor eficiencia.

2.4.3. Algoritmo CSMA-CA

Cuando se emplea este algoritmo los dispositivos anuncian que están listos para enviar paquetes de datos antes de acceder al canal. De esta forma se evita la colisión. Dependiendo de unos ciertos parámetros se da prioridad a uno de los candidatos, que podrá acceder al canal para enviar su paquete de datos. El resto de dispositivos esperarán un tiempo aleatorio (distinto en cada uno de ellos) para volver a intentar acceder al canal.

Se definen dos tipos distintos de algoritmo CSMA-CA según estemos usando el modo balizado o el no balizado. Para el modo balizado se emplea el CSMA-CA ranurado mientras que para el no balizado se utiliza el CSMA-CA no ranurado. En ambos casos, un transmisor sólo puede intentar acceder al medio con una periodicidad concreta (determinada por el tiempo de *backoff*). En el CSMA-CA ranurado, este tiempo de *backoff* debe estar sincronizado con la baliza. Para el caso del CSMA-CA no ranurado, cada transmisor tiene su propio tiempo de *backoff*.

El algoritmo CSMA-CA utiliza tres variables para priorizar el acceso al medio:

- NB (*Number of Backoffs*). Es el número intentos de acceso al canal que un dispositivo lleva acumulados. Cada vez que un dispositivo quiere enviar un nuevo paquete esta variable debe ser inicializada a 0.

- CW (*Contention Window*). Es la longitud de la ventana de contienda. Representa el número de periodos de *backoff* que el canal debe estar sin actividad para que se considere libre y, en consecuencia, poder ser asignado a un dispositivo.

- BE (*Backoff Exponent*). Se emplea para determinar, aleatoriamente, el número de periodos de *backoff* que el dispositivo debe esperar para intentar acceder al canal después de un acceso fallido.

A continuación explicaremos de forma detallada ambos algoritmos. En la figura 2.7 podemos ver su diagrama de bloques.

CSMA-CA ranurado

Este algoritmo se divide en cinco pasos:

1. Inicialización de NB, CW y BE. $NB = 0$ y $CW = 2$. El valor de BE depende de la variable *macBattLifExt*. Si ésta es FALSE entonces $BE = macMinBe$, que por defecto vale 3. Si *macBattLifExt* = TRUE (extensión de vida de la batería) entonces $BE = mínimo(2,\ macMinBE)$.

2. Tiempo de espera aleatorio para intentar acceder al canal. Este tiempo de espera valdrá como máximo $2^{BE} - 1$ periodos de *backoff*. Si $BE = 1$ entonces el tiempo de espera es nulo, con lo que saltamos directamente al paso 3.

3. Evaluación de canal libre (CCA o *Clear Channel Assessment*). El dispositivo escucha el canal en el siguiente periodo de *backoff*. Si el canal no se encuentra en uso, saltamos al paso 5. Si el canal resulta estar ocupado pasamos al paso 4.

4. CW es reiniciado a su valor original (2). NB se incrementa en 1. BE también se incrementa en 1 siempre que no sobrepasemos el valor de *macMaxBE* (constante de valor 5 según el estándar). Si $NB > macMaxCSMABackoffs$ (también de valor 5) se considera que la transmisión ha fallado. Si no, volvemos al paso 2.

5. El canal no está siendo usado. Se disminuye en 1 el parámetro CW. Si CW es 0 entonces se ha superado la ventana de contienda y el canal se considera libre. La transmisión puede comenzar. Si CW no es 0 todavía no hemos superado la ventana de contienda con lo que se vuelve al paso 3.

CSMA-CA no ranurado

Este algoritmo es muy similar al visto en el punto anterior. Los pasos a seguir son:

1. Inicialización de NB, CW y BE. Al igual que en el caso anterior $NB = 0$. En este caso sin embargo $CW = 0$ con lo que no existe ventana de contienda: si se detecta una única vez que el canal no está siendo usado se considera que está libre. Por otro lado, el modo no balizado no soporta el modo *macBattLifExt,* por lo que directamente BE se iguala a $macMinBE$.

2. Idéntico al caso de CSMA-CA ranurado.

3. Idéntico al caso de CSMA-CA ranurado.

4. Idéntico al caso de CSMA-CA ranurado excepto que en este caso no reiniciamos CW a su valor original puesto que estamos utilizando la ventana de contención..

5. Transmisión (como en el caso ranurado).

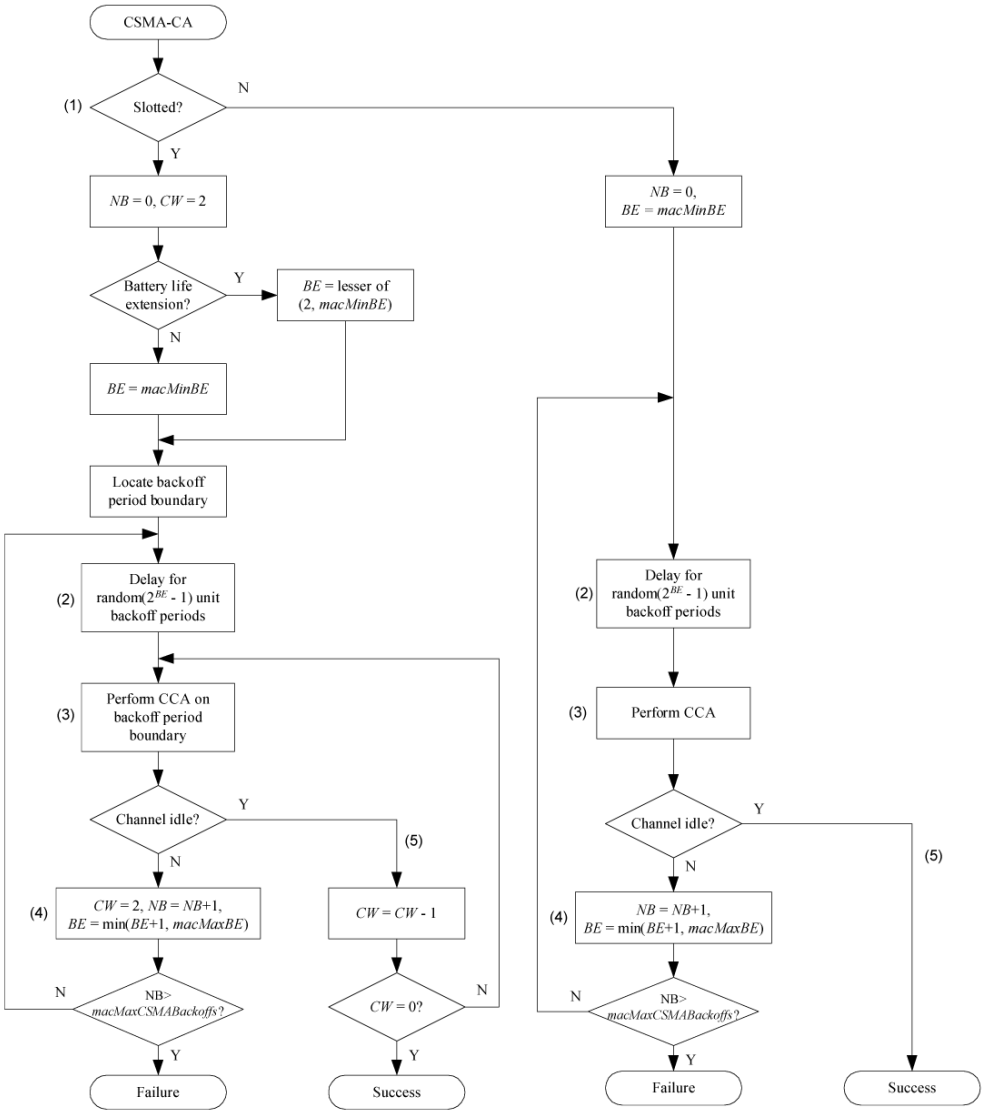


Figura 2.7: Diagrama de bloques del algoritmo CSMA-CA. Fuente: [10]

2.4.4. Inicialización y mantenimiento de PAN

Como hemos mencionado, la capa MAC es la encargada de gestionar la conexión y desconexión de un dispositivo a la red. Para realizar esta tarea, así como la de inicializar la red para el caso del coordinador, la capa MAC se sirve de los siguientes procedimientos:

- **Detección de energía de canal.** Mediante este procedimiento, y a petición de las capas superiores, la capa MAC mide la energía de un canal o lista de canales determinados. Para ello indica a la capa física realizar una detección de energía de canal (ED) para cada uno de los canales elegidos. Mediante este procedimiento el coordinador puede elegir, entre varios, canales aquel con menos tráfico e iniciar la red PAN en él.

- **Escaneo activo de canal.** Permite a un dispositivo localizar a un coordinador que transmita balizas y esté dentro de su rango de alcance. Los *routers* y los dispositivos finales usan este procedimiento a la hora de asociarse a la red. Si el dispositivo se trata de un coordinador, este método se puede usar para detectar las redes PAN dentro de un canal y así seleccionar un identificador distinto a los ya existentes para la nueva red. Un dispositivo que realice un escaneo activo de canal transmitirá periódicamente balizas. Si la red trabaja en modo no balizado, el coordinador responderá a cada una de estas balizas con una nueva baliza. Si la red trabaja en modo balizado, el coordinador ignorará estas balizas y seguirá transmitiendo de forma regular las suyas. Así, en cualquier caso, el dispositivo que realiza el escaneo recibirá una serie de balizas como respuesta a su petición de asociación.

- **Escaneo pasivo de canal.** La única diferencia con el caso anterior es que el dispositivo que realiza el escaneo no transmite baliza alguna. Lo único que hará será escuchar el canal en busca de balizas emitidas por el coordinador. Obviamente, este modo no es adecuado para el modo no balizado.

- **Escaneo de canal por un dispositivo huérfano.** Si el dispositivo ha perdido la sincronización con su coordinador emitirá lo que se conoce como notificaciones de orfandad. Las capas superiores indicarán a la capa MAC por qué canales transmitir dichas notificaciones. El coordinador debe responder a dicha notificación haciendo que la sincronización entre él y el dispositivo huérfano se recupere.

2.4.5. Formato de trama

Se definen cuatro tipos distintos de tramas: formato general de trama MAC, tramas de baliza, tramas de datos y tramas de *ack*.

Estas tramas están compuestas por:

- Cabecera (MHR o MAC *Header*): Se compone por un campo de control, número de secuencia, información de dirección destino y fuente e información relativa a la seguridad utilizada.

- Carga útil. Este campo varía para cada uno de los cuatro tipos de tramas. Las tramas de *ack* no contienen este campo.

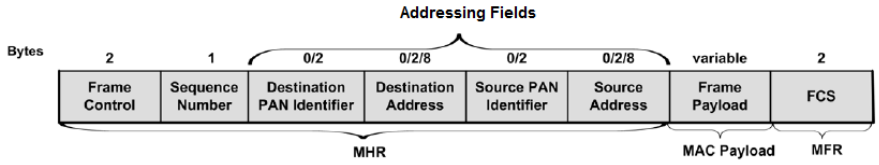
- Secuencia de control (MFR o MAC *Footer*): secuencia de 16 bits conocida como FCS (*Frame Check Sequence*) que no es más que un código CRC (código de redundancia cíclico).

Formato general de trama MAC

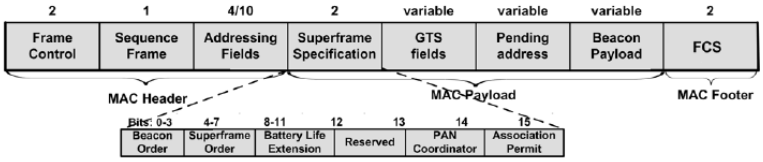
- MHR compuesto por:

  - Control de trama: 16 bits para indicar el tipo de trama, si la trama utiliza un código de seguridad, si se requiere de *ack* por parte del nodo adyacente, etc.

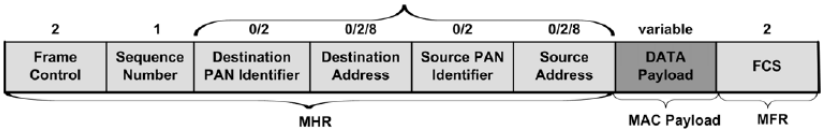
  - Número de secuencia: 8 bits para identificar la trama.



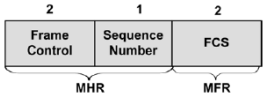
(a) Formato general de trama MAC



(b) Trama de baliza



(c) Trama de datos



(d) Trama de *ack*

Figura 2.8: Distintos formatos de tramas MAC. Fuente: [11]

  - Identificador de PAN destino: 16 bits para identificar el número de PAN con el que nos estamos comunicando. Usamos el valor hexadecimal FFFF para indicar "cualquier PAN".
  - Dirección destino (16 ó 64 bits).
  - Identificador de PAN origen (16 bits).
  - Dirección origen (16 ó 64 bits).

- Carga útil.
- MFR.

Tramas de baliza

El MHR y el MFR son exactamente iguales que en el caso anterior.
En este caso el campo de carga útil tiene el siguiente formato:

- Especificación de supertrama: 16 bits para especificar los parámetros BO, SO, *macBattLifExt,* permiso de asociación, etc.
- Información sobre los campos GTS.
- Direcciones pendientes de ser atendidas (*Pending address*): Lista con las direcciones de los dispositivos pendientes de comunicarse con el coordinador.
- Carga útil: opcional.

Tramas de datos

El MHR y el MFR son exactamente iguales a los casos anteriores.
El campo de carga útil contiene la información pasada por las capas superiores.

Tramas de *ack*

El MHR está compuesto únicamente por los campos de control de trama y de número de secuencia. Este tipo de trama no posee carga útil. El campo MFR es igual que en los casos anteriores.

2.5. Capa de red

2.5.1. Descripción general

La capa de red se define en la especificación de ZigBee [9]. Esta capa es necesaria para gestionar las capas físicas y MAC del estándar 802.15.4 y para proveer de una adecuada interfaz de servicio al nivel de aplicación. Básicamente las tareas que realiza la capa de red son las siguientes:

- Configuración de nuevos dispositivos.
- Inicialización de la red PAN.
- Asociación, reasociación y abandono de una red.
- Adjudicación de direcciones de red.
- Descubrimiento de la topología de red.
- Encaminamiento (*routing*).

Las constantes y atributos utilizadas para configurar la capa de red son listadas en la definición del estándar ZigBee [9] a partir de la página 338. Ejemplos de dichas constantes y atributos son *nwkcCoordinatorCapable, nwkcDefaultSecurityLevel, nwkMaxChildren, nwkMaxDepth, nwkNeighborTable, etc.*

A nivel de red, existen dos tipos de direcciones: direcciones cortas (16 bits) y direcciones largas o direcciones IEEE (64 bits). Cada dispositivo debe tener asignada una dirección IEEE única. No puede haber dos dispositivos que cumplan con la especificación ZigBee y que posean la misma dirección IEEE. Así, esta dirección es asignada en el momento de la fabricación del dispositivo. Por contra, la dirección corta es asignada por la capa de red de forma dinámica. Dentro de una red ZigBee no puede haber más de un dispositivo con igual dirección corta.

2.5.2. Funciones

A continuación explicamos de forma más detallada las funciones principales que realiza la capa de red ZigBee.

Establecer una nueva red

Sólo los dispositivos con capacidad para comportarse como coordinador (constante *nwkcCoordinator-Capable* igual a 1) y no están actualmente asociados a una red pueden realizar este procedimiento.

El primer paso para establecer una nueva red es indicar a la capa MAC que realice una detección de energía de canal para un determinado conjunto de canales o, en su defecto, para todos. Si estamos obligados a trabajar únicamente en un canal, no es necesario realizar dicha detección de energía. Una vez realizada la detección de energía de los canales, se ordenan de mayor a menor calidad, descartando aquellos con un nivel de interferencia demasiado elevado. A continuación se realiza un escaneo activo de canal para cada uno de ellos. Finalmente se elige aquel canal con menor interferencia y en el que trabaje un menor número de redes ZigBee.

Una vez seleccionado el canal se elige un identificador de red menor que el valor hexadecimal 0xFFFF (ya que éste significa "cualquier red") y se indica a la capa MAC dicho valor. A continuación el dispositivo que ha iniciado la red, el coordinador, se autoasigna la dirección corta 0x0000. Por último se indica a la capa MAC que la red ha sido establecida con éxito.

Permitir a los dispositivos unirse a una red

Sólo el coordinador y los *routers* pueden realizar este procedimiento.

La capa de red modificará el valor del atributo *macAssociationPermit* de la capa MAC. Para el caso de que m*acAssociationPermit* sea TRUE el dispositivo en cuestión permitirá a otros dispositivos asociarse a él. La capa de red puede modificar dicho valor de forma indefinida o simplemente durante una cantidad de tiempo determinada.

Descubrimiento de red

Mediante este procedimiento la capa de red indica a la capa superior qué redes ZigBee están operando dentro del rango de alcance del dispositivo en cuestión.

La capa de red solicita a la capa MAC realizar un escaneo activo de canal para un determinado grupo de canales. Una vez realizado la capa MAC responderá indicando en qué canales lógicos están trabajando las redes encontradas, cuáles son sus identificadores (PAN ID) y si permiten o no a otros dispositivos unirse a ellas.

Unirse a una red

El primer paso para unirse a una red es efectuar un "descubrimiento de red", como hemos visto en el punto anterior. Una vez seleccionada la red en la que vamos a trabajar se realiza un listado con los candidatos a "padre" (estos dispositivos serán *routers* y/o el coordinador que estén dentro del rango de alcance del dispositivo en cuestión). Lógicamente descartaremos aquellos que no tengan activado el permiso para unirse ellos. Entre los candidatos restantes seleccionaremos aquellos con menor profundidad (mínima distancia hasta el coordinador); esto quiere decir que el coordinador siempre será prioritario, a continuación los *routers* conectados directamente a él y así en adelante. Si varios candidatos cumplen las mismas condiciones el dispositivo que intenta unirse a la red puede elegir libremente entre ellos.

Durante el procedimiento de unión a la red, el dispositivo debe enviar su dirección IEEE al candidato a dispositivo. El dispositivo padre guardará en su lista de direcciones dicha dirección IEEE, le asociará una dirección corta y, acto seguido, enviará esta dirección corta al dispositivo. En adelante ambos dispositivos, padre e hijo, utilizarán esta dirección corta para comunicarse.

Cuando un dispositivo hijo pierde la conexión con su dispositivo padre, puede solicitar una reasociación con éste. El procedimiento es idéntico al que acabamos de describir excepto por el detalle de que, aunque el dispositivo padre haya cambiado su configuración y ya no acepte a otros dispositivos unirse a él, permitirá asociarse al dispositivo hijo puesto que no se trata de una nueva asociación. Otra opción cuando el dispositivo hijo ha perdido la conexión con el dispositivo padre es solicitar a la capa MAC realizar un "escaneo de canal por dispositivo huérfano". Cuando la capa MAC de un dispositivo huérfano realiza este procedimiento envía su dirección IEEE. Cuando el padre recibe dicha dirección comprueba que dicho dispositivo era su hijo y le responde indicando su antigua dirección corta, con lo que se considera que la conexión se ha recuperado.

Tablas de vecindad

Un dispositivo debe guardar información de cada uno de los dispositivos dentro de su rango de alcance. Esta información es almacenada en la tabla de vecindad. Dicha tabla es de utilidad en diferentes contextos. Primero, cuando un dispositivo intenta unirse a la red utiliza esta tabla para realizar el listado de candidatos a dispositivos padre, como hemos visto en el punto anterior. Segundo, una vez asociado, esta tabla se utiliza para almacenar parámetros relacionados con la calidad del enlace, tipo de relación y demás información referente a los dispositivos dentro de nuestro rango de alcance. La tabla de vecindad debe ser actualizada cada vez que recibimos una trama proveniente del otro dispositivo.

Los parámetros que se almacenan en esta tabla para cada uno de los distintos dispositivos son: dirección IEEE, dirección corta, tipo de dispositivo, relación con dicho dispositivo (el dispositivo es el padre, hijo, hijo asociado a un padre común, antiguo hijo, etc), LQI, número de saltos hasta dicho dispositivo, permiso de asociación, si ha sido seleccionado como candidato a padre, etc.

Mecanismo distribuido de asignación de dirección corta.

Cuando un dispositivo padre (coordinador o *router*) recibe la petición de conexión por parte de otro dispositivo (candidato a dispositivo hijo), el dispositivo padre debe asignarle una dirección corta.

Este algoritmo se basa en la generación de subgrupos de direcciones. El coordinador elige la primera

dirección libre, 0x0000. A continuación, asigna un subgrupo de las direcciones restantes a cada uno de sus hijos con capacidades de *router*. Cada uno de esos hijos tendrá como dirección propia la primera del subgrupo que le ha sido asignado, teniendo las restantes direcciones disponibles para repetir el proceso con los dispositivos hijos que se conecten a él. Una vez asignados estos subgrupos de direcciones a los *routers*, se asignan las direcciones a los dispositivos finales, comenzando por la primera dirección libre.

Para calcular el tamaño de estos subgrupos de direcciones se tienen en cuenta ciertos parámetros de la red definidos por el coordinador: máximo número de hijos que un padre puede tener (*nwkMaxChildren* o C_m), máxima profundidad de la red (*nwkMaxDepth* o L_m) y máximo número de *routers* que un padre puede tener como hijos (*nwkMaxRouters* o R_m). Teniendo en cuenta que d es la profundidad de un determinado dispositivo [9] (número de saltos necesarios por el camino más corto hasta llegar al coordinador, $0 \leq d \leq L_m$) definimos la función $Cskip(d)$ como sigue:

$$Cskip(d) = \begin{cases} 1 + C_m \cdot (L_m - d - 1) & si\ R_m = 1 \\ \dfrac{1 + C_m - R_m - C_m \cdot R_m^{L_m - d - 1}}{1 - R_m} & en\ otro\ caso \end{cases} \tag{2.3}$$

$Cskip(d)$ es el tamaño del subgrupo de direcciones asignado a cada hijo de dicho dispositivo. Si el resultado de la expresión anterior es negativo, se asume que $Cskip(d) = 0$, en cuyo caso el dispositivo no puede tener dispositivos hijo. Formalmente, las direcciones asignadas a los dispositivos finales vienen dadas por la siguiente expresión, donde $Direc_n$ es la dirección asignada al hijo número n y $Direc_{padre}$ es la dirección de su dispositivo padre:

$$Direc_n \quad = \quad Direc_{padre} + Cskip(d) \cdot R_m + n \tag{2.4}$$

En la definición del estándar ZigBee [9] podemos ver un ejemplo de dicho algoritmo. En él, para todos los dispositivos de la red: $C_m = 8$, $R_m = 4$ y $L_m = 3$. En la tabla 2.2 podemos ver el tamaño de los subgrupos de direcciones, $Cskip(d)$, para los distintos valores de profundidad. En la figura 2.9 podemos ver una representación de dicha red.

Profundidad en la red, d	$Cskip(d)$
0	31
1	7
2	1
3 ($= L_m$)	0

Cuadro 2.2: Ejemplo de tamaño de subgrupos de direcciones

Además de este, es importante mencionar que existe otro mecanismo de asignación de direcciones. Su nombre es mecanismo estocástico de asignación de dirección [9]. En él, todas las direcciones (excepto la del coordinador, que sigue siendo la 0x0000) se asignan de manera aleatoria. Únicamente ZigBee PRO soporta el mecanismo estocástico de asignación de dirección.

Abandonar una red

A la hora de abandonar una red se pueden dar dos casos distintos: el padre decide que un hijo debe abandonar la red o el hijo comunica al padre que quiere abandonar la red. En ambos casos el dispositivo padre debe actualizar su tabla de vecindad de forma que el dispositivo eliminado no aparezca en ella.

Encaminamiento

El coordinador de red y los *routers* tienen la opción de crear tablas de encaminamiento. Dichas tablas contienen datos como la dirección corta del destino, el estatus de la ruta, o la dirección corta del siguiente dispositivo en la ruta.

Para calcular la mejor ruta posible se usa un algoritmo que se basa en el "coste" de un camino determinado. El coste se puede calcular teniendo en cuenta el número de saltos necesario y la calidad de los enlaces (usando el indicador LQI).

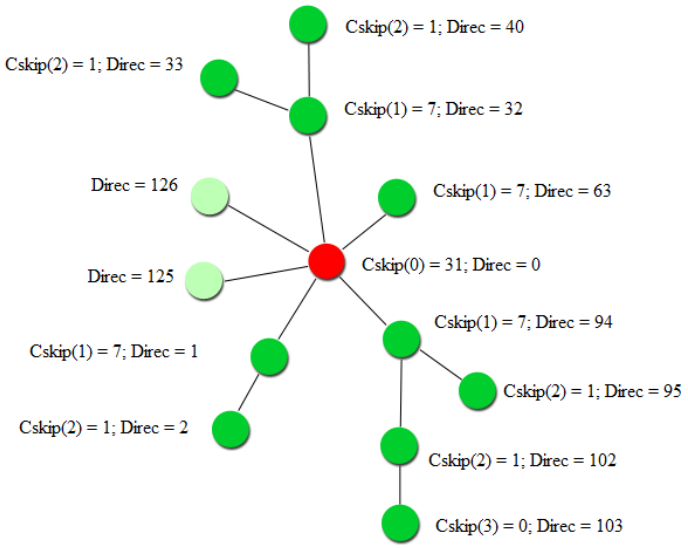


Figura 2.9: Ejemplo de red mostrando el Cskip(d) y las direcciones de red

2.5.3. Formato de trama

El formato general de trama de capa de red se muestra en la figura 2.10.

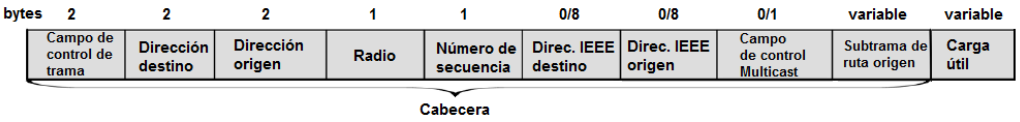


Figura 2.10: Formato general de trama de red

La trama está formada por:

- **Campo de control de trama.** Está compuesto por 16 bits que indican el tipo de trama, la versión del protocolo y si se está empleando seguridad en la transmisión, principalmente.
- **Dirección corta de destino.** El valor 0xFFFF se usa para enviar el paquete a todos los nodos dentro del rango de alcance.
- **Dirección corta de origen.**
- **Radio.** Cada dispositivo que reciba la trama restará uno al valor de esta variable. Así podremos limitar el máximo número de saltos de la trama.
- **Número de secuencia.** Gracias a la dirección origen y al número de secuencia se puede identificar una trama de forma unívoca.
- **Dirección IEEE destino.** Su uso es opcional y es indicado en el campo de control de trama.
- **Dirección IEEE origen.** Su uso es opcional y es indicado en el campo de control de trama.
- **Campo de control *Multicast*.** Su uso es opcional y es indicado en el campo de control de trama. Define parámetros necesarios para la transmisión *multicast*.
- **Subtrama de ruta origen.** Su uso es opcional y es indicado en el campo de control de trama.
- **Carga útil.**

2.6. Capa de aplicación

2.6.1. Descripción general

La capa de aplicación ZigBee está compuesta por la subcapa de soporte de aplicación (APS), el marco de aplicación (*application framework*) y el objeto de dispositivo ZigBee (*ZigBee Device Object* o ZDO). En la figura 2.11 podemos ver una representación de la capa de aplicación y sus diferentes partes.

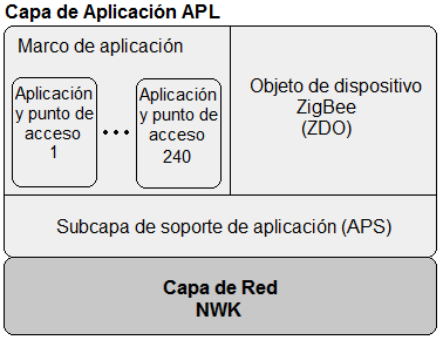


Figura 2.11: Capa de aplicación

En el siguiente subapartardo explicaremos cada una de las partes que componen la capa de aplicación ZigBee. No obstante, antes es necesario explicar el significado de varios términos bastante recurrentes a la hora de hablar de la capa de aplicación:

- **Perfil de aplicación.** Describe el intercambio de mensajes de un conjunto de dispositivos empleados para una determinada aplicación. Por ejemplo, el perfil de aplicación que define un sistema de domótica es distinto al que define un sistema de control de sensores para la industria.
- ***Cluster*.** Es un conjunto de atributos que se utilizan en la comunicación de los distintos dispositivos ZigBee. Por ejemplo, dentro de un sistema de domótica, se puede definir un *cluster* para el control de luces, otro para el control de temperatura, etc.
- **Punto de acceso (*endpoint*).** Dentro de un mismo dispositivo, se pueden definir varios puntos de acceso. Cada uno de estos puntos de acceso gestiona el funcionamiento de una aplicación diferente. Siguiendo con el ejemplo del sistema de domótica, un *router* que funcione como control remoto puede tener un punto de acceso para el manejo del sistema de alarmas, otro para el control de luces, otro para el de temperatura, etc.
- **Vínculo (*binding*).** Es una conexión lógica entre un punto de acceso origen y uno o varios de destino. Sólo se puede realizar un vínculo entre puntos de acceso que compartan el mismo *cluster*. Ya que un dispositivo puede poseer varios puntos de acceso, también puede soportar varios vínculos.

2.6.2. Subcapas

Subcapa de soporte de aplicación (APS o *Application Sublayer*)

La subcapa de soporte de aplicación proporciona una interfaz de comunicación entre la capa de red y la capa de aplicación. Las tareas que realiza esta capa son:

- Genera la PDU a nivel de aplicación.
- Una vez los puntos de acceso de dos dispositivos están vinculados, la capa APS es la encargada de gestionar el intercambio de mensajes.
- Mejorar la fiabilidad de la capa de red. Por ejemplo, se puede definir una confirmación *ack* a nivel de aplicación.
- Rechaza mensajes recibidos por duplicado.

Marco de aplicación (AF o *Application Framework*)

El marco de aplicación es el entorno en el cual se gestionan las diferentes aplicaciones. Cada una de dichas aplicaciones está relacionada a un punto de acceso distinto.

Se permiten hasta 240 aplicaciones distintas dentro de un mismo dispositivo. El punto de acceso 0 está asignado al nivel ZDO. El rango de 241-254 está reservado para uso futuro. Por último, el punto de acceso 255 es usado para la comunicación *broadcast* con todas las aplicaciones dentro del marco de aplicación.

Esta capa es la encargada de definir tanto el perfil de aplicación como los diferentes *clusters*. Cada *cluster* se caracteriza por un identificador propio (*cluster* ID).

Objeto de dispositivo ZigBee (ZDO o *ZigBee Device Object*)

Este nivel satisface necesidades comunes a todas las aplicaciones dentro del marco de aplicación. Las tareas que realiza son:

- Inicializar las capas APS y de red.
- Definir el rol del dispositivo dentro de la red (coordinador, *router* o dispositivo final).
- Gestiona los vínculos entre puntos de acceso.
- Asegura una comunicación segura entre dispositivos.

Como hemos dicho anteriormente, el ZDO se encuentra en el punto de acceso 0.

2.6.3. Formato de trama

El formato general de trama de capa de aplicación se muestra en la figura 2.12.

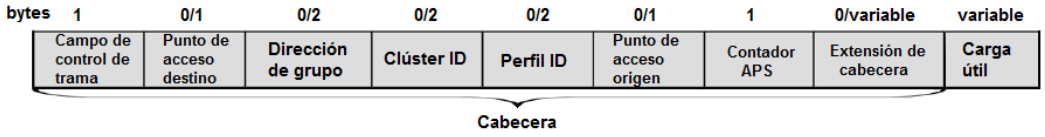


Figura 2.12: Formato general de trama de aplicación

La trama está compuesta por:

- **Campo de control de trama.** Está formado por 8 bits que indican el tipo de trama, si se utiliza seguridad, si estamos empleando la extensión de cabecera y si se requiere confirmación *ack* a nivel de aplicación.
- **Dirección del punto de acceso destino.**
- **Dirección de grupo.** Si se usa direccionamiento de grupo la trama no debe incluir la dirección del punto de acceso destino. Todos los puntos de acceso asociados a la dirección de grupo en cuestión recibirán esta trama.
- ***Cluster* ID.** Identificador de *cluster*.
- **Perfil ID.** Identificador del perfil de aplicación.
- **Dirección del punto de acceso origen.**
- **Contador APS.** Se usa para evitar la recepción de tramas duplicadas.
- **Extensión de cabecera.** Extiende la funcionalidad de la cabecera.
- **Carga útil.**

2.7. Seguridad

Desde el punto de vista de la seguridad, las redes inalámbricas son altamente vulnerables debido a que no se requiere tener acceso a un medio cableado para participar en las comunicaciones. ZigBee/802.15.4 está orientado hacia un mercado en el que el bajo consumo y el bajo coste son requisitos esenciales. Debido a la restricción en lo que al coste se refiere, los dispositivos que trabajan con esta tecnología se enfrentan a una limitación computacional evidente. Esto quiere decir que los sistemas de seguridad son mas difíciles de implementar en sistemas que trabajan bajo ZigBee/802.15.4. Además, a la hora de implementar mecanismos de seguridad para ZigBee/802.15.4 se ha tenido en cuenta que los dispositivos no tienen por qué tener acceso a una base de datos de forma continua. De esta forma, los distintos dispositivos deben ser capaces de gestionar los mecanismos de seguridad por sí mismos. Por último, es importante destacar que ZigBee/802.15.4 está orientado hacia en envío de pequeños paquetes de información, por lo que los mecanismo de seguridad utilizados no deberían aumentar en exceso el tamaño de la cabecera

En los estándares ZigBee [9] y 802.15.4 [10] se definen mecanismos de seguridad para las capas MAC, de red y de aplicación. El mecanismo criptográfico utilizado se basa en el uso de claves simétricas. La clave en cuestión debe ser generada por las capas superiores. El mecanismo criptográfico debe ser capaz de asegurar:

- Confidencialidad de los datos o privacidad. Esto quiere decir que la información transmitida sólo de llegar a los dispositivos asociados a la red.
- Autentificación de los datos. Se debe asegurar que la información recibida proviene de un dispositivo válido (en otras palabras, no ha habido intromisión en las comunicaciones).
- Detección de información duplicada. Los paquetes de información deben de ser recibidos únicamente una vez.

De acuerdo al estándar [10] la clave utilizada puede ser conocida por los dos dispositivos que participan en la comunicación o por un grupo de dispositivos. Si utilizamos una única clave para un grupo de dispositivos ganamos en simplicidad pero, por contra, estamos desprotegidos ante el ataque de un dispositivo de la propia red.

ZigBee/802.15.4 usa el modo CCM*, una variante del modo CCM (*Counter with CBC-MAC*). CCM* usa bloques de encriptación basados en el algoritmo AES-128 (*Advanced Encryption Standard*). CCM* es un modo simétrico, lo que quiere decir que tanto el dispositivo que transmite como el que recibe usan la misma clave para descifrar el mensaje.

En caso de que la seguridad esté habilitada, el coordinador es normalmente el que realiza las funciones de centro de seguridad, permitiendo o no la asociación de un nuevo dispositivo. El centro de seguridad debe actualizar periódicamente la clave utilizada, cambiándola por una nueva si lo estima necesario. Para ello, distribuye la nueva clave encriptada con la antigua. Después comunica a todos los dispositivos que a partir de ese momento deben usar la nueva clave.

El centro de seguridad, que como hemos dicho suele ser el coordinador pero que también puede ser un dispositivo expresamente dedicado a ello, realiza las siguientes funciones:

- Autentificar dispositivos que quieren unirse a la red.

- Generar y distribuir nuevas claves.
- Habilitar el uso de seguridad punto a punto entre dispositivos.

ZigBee/802.15.4 usa tres tipos de claves:

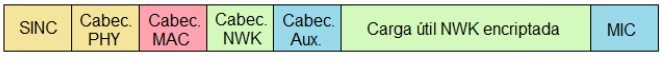
- **Claves de enlace.** Sirven para dotar de seguridad las comunicaciones punto a punto a nivel de aplicación. Sólo lo dos dispositivos que participan en la comunicación conocen esta clave.
- **Claves de red.** Esta clave se usa para la seguridad a nivel de red. Todos los dispositivos dentro de una misma red deben compartir esta clave.
- **Claves maestro.** Esta clave es utilizada por dos dispositivos en el inicio de la comunicación para generar la clave de enlace. La clave maestro no se usa para encriptar tramas.

Se definen dos tipos de seguridad:

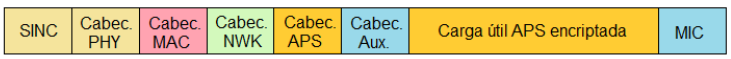
- **Modo de seguridad estándar.** En este modo la lista de dispositivos pertenecientes a la red y las diferentes claves pueden estar almacenadas dentro de cada uno de los distintos dispositivos. El centro de seguridad solamente se encarga de mantener una clave de red común y de controlar las políticas de admisión. Así, el centro de seguridad no necesita una gran memoria para almacenar los datos relacionados con la seguridad de la red.
- **Modo de seguridad avanzado.** En este caso el centro debe almacenar tanto las claves como el listado de dispositivos, así como de controlar las políticas de admisión. Conforme crezca el número de dispositivos asociados a la red, aumentarán los requerimientos de memoria del centro de seguridad. Solamente ZigBee PRO soporta este modo de seguridad.

Las diferentes capas añadirán una cabecera auxiliar en el caso de que utilicemos un modo seguro de transmisión. Además, se debe incluir un código de integridad (MIC o *Message Integrity Code*) a continuación de la carga útil. Figura 2.13.

(a) Capa MAC



(b) Capa de red



(c) capa de aplicación

Figura 2.13: Formato de trama con seguridad habilitada

2.8. Implementaciones comerciales

Algunos de los principales fabricantes de dispositivos electrónicos basados en ZigBee/802.15.4 son:

Atmel (www.atmel.com)

Esta empresa ofrece dispositivos basados en la combinación de su serie de microcontroladores conocida como AVR y un sistema radio especialmente diseñado para 802.15.4. Atmel ofrece dos gamas de dispositivos:

- Series AT86RF230 y AT86RF231. Ambas trabajan a 2,4 GHz.
- Serie AT86RF212. Esta serie trabaja en las frecuencias de 868 y 915 MHz y cumple con el estándar 802.15.4c. Dicho estándar es una variante del 802.15.4 que especifica el uso de la banda ISM en China. Con la comercialización de esta serie Atmel pretende expandir su cuota de mercado en dicho país.

El entorno de desarrollo utilizado para la programación de los microcontroladores es AVR Studio.

Atmel también dispone de *kits* de desarrollo/evaluación, como ATAVRRZRAVEN, muy útiles para desarrolladores en las primeras etapas de diseño.

Ember (www.ember.com)

Ember se centra casi exclusivamente en la fabricación de dispositivos relacionados con ZigBee/802.15.4. Los productos básicos que ofrece son:

- EM250 *SoC*. Sistema que engloba tanto el microprocesador como el transceptor. Trabaja a 2,4 GHz.
- EM260 *Co-Processor*. Este chip combina un transceptor, que también trabaja a 2,4 GHz, con un microprocesador de capacidad limitada. Tiene una conexión SPI (*Serial Peripheral Interface*)/UART (*Universal Asynchronous Receiver-Transmitter*) para poder conectar cualquier microprocesador a dicho sistema. De esta forma se permite al diseñador implementar de forma independiente la parte más alta de la capa de aplicación (marco de aplicación).

Las capas de red y de aplicación de dichos dispositivos trabajan usando EmberZNet PRO, una versión propia del estándar ZigBee.

Ember también ofrece *kits* de desarrollo basados bien en EM250 o en EM260.

Freescale (www.freescale.com)

En relación con ZigBee/802.15.4, Freescale ofrece las siguientes gamas de productos:

- MC1320X *RF Transceiver*. Transceptor basado en 802.15.4. Posee una conexión SPI para poder conectarlo a cualquier microprocesador.
- MC1321X *System in Package*. Combina un transceptor MC13202 con un microprocesador de freescale (MC9S08GT).

- MC13224 *Platform in Package.* Combinación de transceptor 802.15.4 y microcontrolador centrada en la optimización energética del transceptor.

La versión ZigBee de Freescale es conocida como BeeStack. Respecto a la capa MAC, Freescale ofrece la posibilidad de usar el protocolo SMAC (simple MAC), una versión propia y simplificada de la capa MAC 802.15.4.

Los *kits* de desarrollo que ofrece son 1321X ó 1322X *Development Kits.*

Jennic (www.jennic.com)

Jennic oferta los siguientes chips:

- JN5148.
- JN5139.
- JN5121.

Los tres están compuestos por un transceptor y un microprocesador. JN5148 es el producto más reciente y el que ofrece unas mejores características.

Jennic tiene una versión propia de la capa de red conocida como JenNet.

Dispone de un *kit* de desarrollo para principiantes, *JN5139 IEEE802.15.4/JenNet Starter Kit,* y de una versión más profesional, *JN5139 IEEE802.15.4/JenNet Evaluation Kit.*

Texas Instruments (www.ti.com)

Texas Instruments (TI) ofrece varias gamas de productos ZigBee/802.15.4, todas ellas trabajando a 2,4 GHz:

- CC2430 y CC2431. Combinación de transceptor y microcontrolador en un único chip. Trabaja usando Z-Stack, la versión de ZigBee de TI. Los *Kits* de desarrollo asociados son CC2430DK y CC2431DK.
- CC2480. Coprocesador ZigBee con conexión SPI/UART. Está pensado para trabajar conectado a un microprocesador. El coprocesador ZigBee maneja el sistema radio y en él se integran las capas física, MAC, de red y parte de la de aplicación. El *Kit* de desarrollo asociado es eZ430-RF2480.
- CC2420 y CC2520. Transceptor 802.15.4 pensado para trabajar conectado a un microprocesador MSP430 de TI. El *Kit* de desarrollo asociado es CC2520DK.

TI también brinda la oportunidad de trabajar con una capa MAC propia conocida como TIMAC.

El entorno de desarrollo utilizado para la programación de los microcontroladores es IAR *Embedded Workbench* [31].

El listado de empresas expuesto no es exhaustivo y sólo pretende servir como visión general del mercado de componentes electrónicos ZigBee/802.15.4. Por supuesto, otras muchas empresas dedican sus esfuerzos al desarrollo y comercialización de este tipo de dispositivos. Ejemplos de ello son Microchip Technology, STMicroelectronics, Uniband Electric Corporation, ZMD, etc.

De entre todas las opciones comerciales posibles hemos elegido el *kit* de desarrollo eZ430-RF2480 de TI para la realización de nuestro proyecto. En primer lugar TI es una empresa con gran proyección de futuro dentro del campo de los componentes electrónicos orientados a ZigBee/802.15.4. En segundo lugar, eZ430-RF2480 nos ofrece todo lo necesario para realizar nuestro estudio sobre el consumo de corriente en nodos 802.15.4 de forma más simple y rápida que utilizando otros *kits* de desarrollo.

En el siguiente capítulo expondremos en profundidad las componentes *hardware* y *software* del *kit* eZ430-RF2480.

Apéndice B

Texas Instruments® SimpliciTI™ Documentation

B.1. Especificación del Protocolo

SimpliciTI: Simple Modular RF Network Specification

Author: Larry Friedman

Texas Instruments, Inc.
San Diego, California USA

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Version	Description	Date
0.99	Initial release to BDSNet Team	04/25/2007
1.00	Version for wider release with changes based on reviews and early implementation.	05/09/2007
1.01	• Modifications to some network application frame formats • Join API doesn't exist anymore but was included in pseudo-code example. • Add poll to Mgmt port messages.	05/21/2007
1.02	• Change references from SMRFNet to SimpliciTI	06/12/2007
1.03	• Update to reflect addition of callback support.	07/02/2007
1.04	• Add number-of-connections byte to Join frame to support AP data hub scenario • Enhance description of callback in support of AP data hub scenario	08/01/2007
1.05	• Fix error in frame content Figure for client side Join frame	01/02/2007
1.06	• Frame drawing more generic • Add transaction ID to NWK application frames • Poll frame needs SimpliciTI address in addition to port • Frequency application frames defined • Ping application payload now fixed length. • SMPL_LinkListen() now a timed wait.	04/04/2008
1.07	• Add Unlink support	08/08/2008
1.08	• Changes to frame format for auto acknowledgement • Extended API • Added frame objects for Link session (Security)	02/27/2009
1.09	• Section 4.11, update the frequencies that SimpliciTI supports.	03/24/2009

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

TABLE OF CONTENTS

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

LIST OF FIGURES

LIST OF TABLES

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

1. Introduction

This document presents the specification for a simplified RF network solution. This solution is intended to support quick time-to-market for customers wanting a wireless solution of low power, low cost and low data rate networks without needing to know the details of the wireless network support.

This document specifies a simplified approach to small low data rate wireless network solution

2. Abbreviations and terminology

AP	Access Point: one of the 3 SimpliciTI device types.
API	Application Programming Interface
CCA	Clear Channel Assessment; part of listen-before-talk discipline
Customer	The target of this solution: the entity that will use SimpliciTI in their products.
ED	End Device: one of the 3 SimpliciTI device types.
End User	The Customer's customer
LBT	Listen-before-talk: the means used to arbitrate use of the radio spectrum in the network
LLC	Logical Link Control
MAC	Medium Access Control
MCU	Microcontroller Unit
RAM	Random Access Memory
RE	Range Extender: one of the 3 SimpliciTI device types.
SoC	System-0n-Chip
UI	User Interface

3. Sample Applications

The following are a few sample applications that exemplify the potential market for devices supported by SimpliciTI.

3.1 Alarms and security

These applications cover such devices as glass-break sensors, occupancy sensors, door locks, carbon monoxide and light sensors, etc.

3.2 Smoke detectors

Similar to alarm applications, the smoke detector application is prevalent enough so that it deserves its own category. This application can include cascaded smoke detectors in which the activation of a single device propagates to the activation of all devices co-located to efficiently spread an alarm.

3.3 Automatic Meter Reading (AMR)

Some AMR environments are appropriate for SimpliciTI implementations. In particular, this would include meters that are not powered such as gas or water meters.

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

3.4 Home automation

This can include device control such as garage doors, appliances, environmental devices, etc.

4. Requirements

4.1 Low Power

SimpliciTI is intended to support low powered devices when the devices are not mains-powered. These devices will typically be battery powered low data rate devices arranged in a simple, limited topology.

R1. Power management of the MCU shall be accomplished at the application level using native MCU resources.

R2. Power management of the radio shall be available to the application via function calls (see R7).

4.2 Low Cost

There are two configurations to be supported: the radio-MCU solution and a SoC solution that integrates MCU and radio.

R3. To keep the cost low the target MCU MSP430 core shall be 8K flash and 512 bytes RAM or 4K flash and 256 bytes of RAM depending on device type (see R10).

R4. The target SoC (8051 core) target shall be less than 16K flash and 1K RAM.

4.3 Simple Implementation and deployment

The SimpliciTI-based network should be easy to develop and easy to deploy. The simplified development environment should have a simple API that enforces details of the RF communication without complex configuration by developers.

The simplified RF environment should also make it easy for customers to develop applications that make the solution easily implemented in the field by end users.

R5. Radio and network configuration shall be encapsulated and hidden from the application layer and shall be driven by build-time configuration possibly assisted by a tool such as SmartRF Studio.

R6. The API set for messaging in this environment shall be of the open/read/write/close variety.

R7. There shall be access run-time configurability using an **`ioctl()`**-like method.

R8. Linking application peers shall be of a **`link()/listenLink()`** variety.

4.4 Target radio configurations

R9. The order for releases of code for target radios for this project shall be:

1. CC1100/CC2500 (Sub 1 GHz/2.4 GHz radio with MSP430)
2. CC1110/CC2510 (Sub 1 GHz/2.4 GHz radio in 8051 core SoC)
3. CC2430/CC2420 (DSSS radio with and without 8051 core SoC)

4.5 Limited network device types

The devices described are logical device types. They may or may not map to unique physical devices. A single physical device may function as more than one logical device.

R10. Only 3 device types shall be defined. There shall be an End Device type (mandatory), an Access Point device type (optional), and a Range extender device type (optional).

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

4.5.1 Access Point

R11. An Access Point (AP) shall be always on and presumed to be mains powered possibly with battery backup. Only 1 AP per network is permitted.

R12. APs shall have the following functions not all of which may apply on a given network:

1. Network address management
2. Store-and-forward messages on behalf of sleeping Rx devices

R13. An AP may contain sensor/actuator (End Device) functionality.

R14. An AP may contain Range Extender functionality.

4.5.2 Range Extender

R15. The Range Extender (RE) shall be always-on and presumed to be mains powered possibly with battery backup. The device retransmits each unique frame it receives.

R16. A Range Extender may contain sensor/actuator (End Device) functionality.

4.5.3 End Device

R17. The End Device (ED) realizes the application layer functionality.

R18. An End Device may or may not be always on.

R19. End Devices may be RxTx devices or they may be Tx-only devices.

R20. Unless configured externally, Tx-only devices:

1. Transmit on a preset single or sequence of frequencies or channels.
2. Use a preset encryption key to encrypt and decrypt messages.

Tx-only devices may have switches, buttons, or other UI opportunities for simple network configuration.

4.6 Topology

R21. There shall be two logical messaging configurations: star and direct peer-to-peer (device to device).

4.6.1 Star

When an AP provides store-and-forward support for sleeping Rx devices the AP acts like a hub in a Star network topology. If all Rx devices are always on the AP provides no store-and-forward support but can provide network management support and Range Extender functionality.

4.6.2 Peer-to-peer

Rx devices that are always on will receive frames directly from the source device, possibly through a Range Extender. These are considered peer-to-peer relationships. It is possible that the network has no AP in which case the topology is entirely peer-to-peer. Even if all messages are broadcast, from the application perspective messages are coming from a peer.

4.7 Routing

R22. Explicit routing shall not be supported.

R23. Rx devices that are not always on may receive data by polling the AP possibly through a Range Extender. The variety of low-power modes offered by the combination of MCU and radio may support interrupt-driven schemes as well.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

R24. Rx devices that are always on shall receive data directly from the source possibly through a Range Extender.

4.8 Security

4.8.1 Encryption

R25. Where hardware encryption is supported it shall be available for use by the customer. If hardware encryption is not available a default software encryption solution shall be available to the customer.

R26. Encryption key distribution shall be part of the network management support optionally provided by the AP. If there is no AP or a network member is a Tx-only device key settings will be accomplished using a default build-time scheme or possibly be using external settings imposed by the end user.

R27. The encryption solution shall use symmetric keys.

4.8.2 Rogue devices

R28. Rogue devices, malicious or not, shall be prevented from disrupting a network. There shall be defense against such interference as replaying of frames.

4.9 Medium access

R29. Access by a device to transmit will be managed by listen-before-talk procedure. The procedure shall follow the European (ETSI) specification for this algorithm.

4.10 Frequency agility

Frequency agility is intended to support robustness by providing a means to change frequency when a specific frequency is noisy or otherwise ineffective. It is realized in the form of channel migration. Channel migration not intended to be used also to spread energy over the spectrum to comply with FCC Part 15 requirements. It is not intended that frequencies change in any rapid manner, for example, packet-to-packet.

R30. The network shall support migrating to alternate frequencies if an existing frequency offers too much radio interference.

4.11 Radio parameters

R31. Each network shall support up to 250 kbps.

R32. The following frequencies shall be supported: 480, 868, 915, 955 MHz (sub-1 GHz) and 2.4 GHz.

4.12 Range extension without transmit loops

R33. Frame content shall contain hints that prevent looping.

R34. Range extension shall be limited to 4 hops from the message source to the message destination.

R35. The number of Range Extenders shall be limited to 4 (four) in a network.

R36. A Range Extender will not resend frames that it receives from another Range Extender

4.13 Battery-only networks

R37. Battery-only networks shall be supported. This implies that there will be no Range Extenders and no AP in such a network.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

4.14 Interoperability

R38. Interoperability among versions of SimpliciTI with different feature sets shall not be required. Interoperability may be supported as a side effect of released versions but it is not required and therefore not guaranteed.

5. Overview of SimpliciTI

SimpliciTI is intended to support customer development of wireless end user devices in environment in which the network support is simple and the customer desires a simple means to do messaging over air.

The protocol is oriented around application peer-to-peer messaging. In most cases the peers are linked together explicitly. However, SimpliciTI will support scenarios in which explicit linking between pairs of devices is neither needed nor desired. An example if this is the smoke alarm scenario in which there are multiple sensors/alarms and the intent is to propagate the alarm throughout the network.

In this case an activated alarm would send a broadcast message to any and all nearby alarms so that the alarm signal could be propagated throughout the network of alarms. Since explicit linking has not occurred in this scenario the Customer must take care to choose the default access token so that interference from non-network sources is not likely.

From the Customer's development perspective the API provides the means to initialize the network, link devices, and send messages.

5.1 Module components

The basic stack will support simple RF messaging with no additional features. As described in the figure below there are modules (features/functions) that can be added to the basic stack in various combinations.

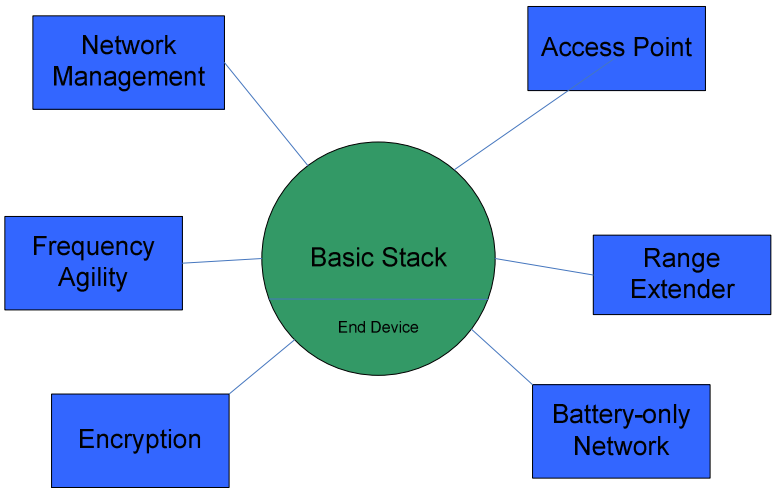


Figure 1: SimpliciTI Module components

A brief description of each component follows.

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

5.1.1 Basic stack

5.1.2 Encryption

The encryption choices are currently hardware or software. The CC12100/CC2500 radios do not have native support for hardware encryption so on platforms using these radios the encryption will be in software.

When encryption is enabled all fields except the address and encryption context fields are encrypted. Though this means that Range Extenders need to decrypt the frame to know whether to repeat it this will prevent rogue devices from storming the network with bogus frames leveraging off the Range Extenders.

5.1.3 Frequency agility

This capability allows the network to migrate to a different frequency if the existing frequency is noisy or otherwise compromised. It will be driven by a frequency table that is populated at build time.

Devices that can receive packets can detect that they are on the incorrect frequency by not receiving an acknowledgment after sending and resending a frame. The sender then steps through the frequency table until the acknowledgment is received. This is the scenario encountered by new devices trying to join a network and sleeping devices that awaken after a migration has occurred.

Changing frequency may also be actively imposed. In networks with an AP the AP will announce the change with a broadcast message. In networks without an AP there may be an external hardware announcement such as a switch or button to signal each device.

Tx-only devices must always send and resend on all frequencies in the table.

Detection of the need to change frequencies is not within the scope of this specification.

5.1.4 Network management

This is the domain of the AP and consists of functions such as store-and-forward functionality for sleeping devices, encryption key management, and frequency agility management.

5.1.5 Access Point

Access points can support End Devices.

Access Points will repeat frames but they set the Access Point Sender Type (see Section 6.5.5). When replaying a frame the hop count is decremented.

Access Points realize Network management functions.

5.1.6 Range Extender

Subject to anti-congestion restrictions enforced by the hop count Range Extenders replay all received packets unless either the RE itself is the destination of the frame (e.g., a Ping) or the Range Extender Sender Type is set in the received frame(see Section 6.5.5). APs will also replay frames but since the Access Point Sender Type is set in these frames Range Extenders are permitted to replay frames from APs. When replaying a frame the hop count is decremented.

5.1.7 Battery-only network

Because APs are always powered battery-only networks do not have APs. In battery-only networks the presumption is that all devices are sleeping devices. Since there is no store-and-forward capability the receipt of frames depends on retransmissions by the sender. The proper balance between the frequency of a receiving device awakening, the length of time the device remains listening, and the frequency with which the sending device retransmits must be in place.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

5.2 Architecture overview

SimpliciTI software conceptually supports 3 layers as shown in Figure 2. The Application Layer is the only portion that the customer needs to develop. The communication support is provided by a simple set of API symbols used to initialize and configure the network, and read and write messages over air.

The architecture does not strictly follow the OSI Reference model.

a. There is no formal **`PHY`** or Data Link (**`MAC/LLC`**) Layer. The data are received directly from the radio already framed so the radio performs these functions.

b. Since the security support is here implemented as a peer of the Network layer there is no Presentation Layer where this is function is formally implemented in the OSI model.

c. If reliable transport is required that must be implemented by the application so there is no Transport layer.

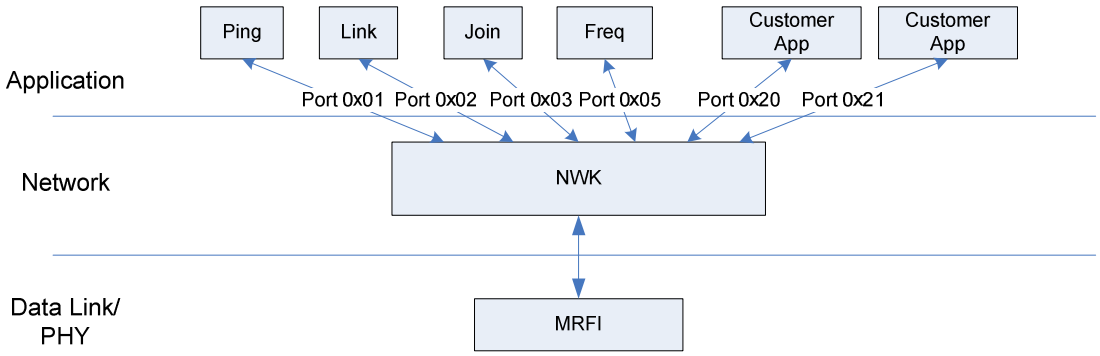


Figure 2: SimpliciTI Architecture

5.2.1 Application Layer

The customer develops the application as the implementation of sensor/actuator interactions with the environment. Using the SimpliciTI API the application can send/receive messages to/from an application peer on another device.

Management of the network itself is supported by SimpliciTI network "applications" as shown above. Each has its own application protocol as would any customer application. These applications can be easily extended and modified as needed. They are described in detail in Section 7.

5.2.2 Network Layer

The network layer actually spans the boundaries of the standard OSI model, as it collapses and hides functionality from the application.

5.2.2.1 Encapsulated Network parameters

Network setup is driven by build-time configuration schemes. These schemes may include code generation tools that generate static objects, header files, and manifest constants. Run time adjustment of some of these may be accessible from application via an **`ioctl()`**–like interface.

Network parameters can include:

1. Base frequency and frequency spacing
2. Number of frequencies supported (for frequency agility table)
3. Modulation method and data rate and other general radio parameters

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

4. Default and generated network encryption keys
5. Number of store-and-forward messages to hold
6. Device address
7. Repeat rates on Tx-only devices
8. Join and link tokens

5.2.3 Minimal RF Interface (MRFI)

This layer abstracts what is basically a frame read/write interface to the radio. Different radios supported by SimpliciTI require different implementations but the basic interface offered to the network layer is the same for all radios.

Different radios offer different levels of support for typical Data Link and PHY layer responsibilities. MRFI encapsulates these differences.

5.2.4 Board Support Package (BSP)

As a convenience there will be some minimal support for various target MCUs. In the form of a BSP (not shown in architecture drawing) the following will be abstracted:

1. SPI interface to the radio (when appropriate)
2. Interrupt management via GPIO connections to support asynchronous notifications from the radio (when appropriate)
3. LED and button support using GPIO lines.

5.3 Topology

Below are drawings of examples of the topologies supported by SimpliciTI. In each of the following End Device 1 (ED1) is sending data to End Device 2 (ED2). The following legend applies to the line formats:

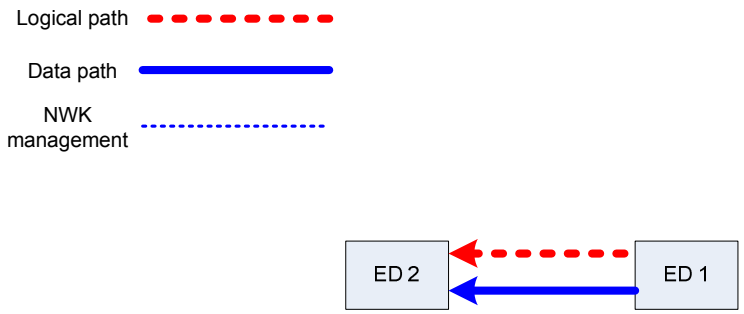


Figure 3: Direct peer-to-peer

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

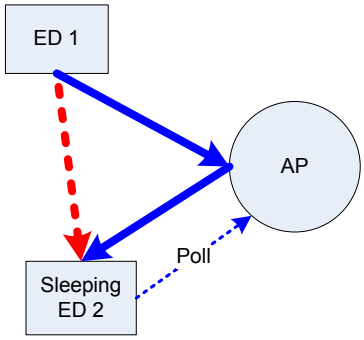


Figure 4: Store-and-forward peer-to-peer through Access Point

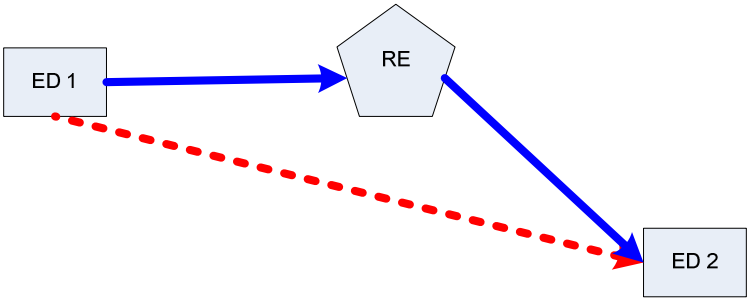


Figure 5: Direct peer-to-peer through Range Extender

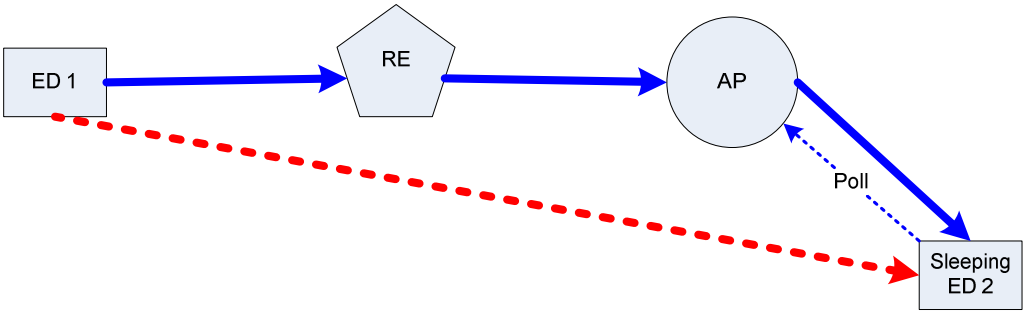


Figure 6: Store-and-forward peer-to-peer through Range Extender and Access Point

5.4 Network discipline

All frames are sent to and from applications on top of the **NWK** layer. There are both user applications and **NWK** applications. The user application frames are peer-to-peer user application messages. Other than fixed network information overhead such as destination address, frame length, etc., the packet's payload is application data.

The network management is accomplished by **NWK** applications whose frames are also peer-to-peer. Network management includes items such as device linking, radio frequency management, security key management, and network membership management. See Section 6.5.4 for more detail.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Frame acknowledgement is the responsibility of the application, be it a user application or a **NWK** application. The acknowledgment is part of the application peer-to-peer discipline. So, for example, the acknowledgment to a **Ping** frame is the echoed **Ping** frame in response.

User applications are connection based. Connections are established as bi-directional (but see Section 7.2.2.1).The connections are established when the devices are linked.

The **NWK** applications are connectionless. Some of these applications implement acknowledgements and some do not. See Section 7 for more detail.

5.5 Network formation

The following will be the procedure for an AP when the initialization call is made (forward references are made here to Section 6.5.4):

- Set the network frequency to the default frequency by broadcasting a **Freq** frame on all tabled frequencies.
- Set the encryption key by broadcasting a **Security** frame on the network frequency.
- Support **Join** frames sent by newly arriving devices.
- Support store-and-forward for joining devices that can receive frames but are not always on.

5.5.1 Address namespace usage

The address namespace consists of all 4-byte values except those that map to the radio's reserved broadcast address.

5.6 Power management

The SimpliciTI simple RF network is intended to support low power devices. In the dual chip solution both the radio stage and the MCU stage may have power managed independently.

It is the responsibility of the application to manage power on the MCU in the dual chip solution. The radio power management can be derived from the MCU power management by using the **`ioctl()`**-like interface provided in the API (See Section 8.8).

5.7 Transmit-only (Tx-only) devices

Transmit-only devices have the following behaviors:

- They use default network key. They may use another key if Customer provides some means for the stack to detect and implement the key such as an external switch.
- They use retransmission to ensure packet propagation since they cannot receive acknowledgments.
- There are two ways to handle frequency migration on Tx-only devices:
    - They transmit each frame on all tabled frequencies since they cannot receive acknowledgments.
    - Frequency changed by external intervention such as a switch or button.
- When a Tx-only devices links to another device the link token must be valid.
- Other network infrastructure will not acknowledge frames from Tx-only devices.

5.8 Network byte order

Multi-byte number objects are to be represented in little endian format. This convention is followed for number objects contained in network application payloads.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

6. Frame structure

Frames consist of 3 logical portions:

- the portion processed by the **PHY/MAC** layers
- the portion supporting network management implemented in the **NWK** layer
- the portion representing the application payload supported in the application layer.

6.1 PHY/MAC frame portion

This portion contains the parts of the frame processed by the hardware. For the CC1100/CC2500 radios this consists of the preamble bits and the sync bits.

6.2 NWK frame portion

This portion of the frame is processed by firmware in the **NWK** layer. The fields in this portion are for network control and specify such parameters as frame type, encryption status, hop count, sequence number, etc.

6.3 Application frame portion

This is the encapsulated application data payload that is delivered to the application as a result of a receive API call for user applications. For **NWK** applications the application portion of the frame is processed as part of the receive thread.

6.4 Address fields

SimpliciTI does not have separate address fields for each layer as some protocols do. The 4-byte SimpliciTI address applies in general. This leads to an architectural ambiguity as to which layer actually "owns" the address.

The solution for SimpliciTI is that the network layer owns the address but control of the address fields themselves rests with MRFI. This is because the encapsulated radio knowledge determines how the fields are actually mapped to the radio frame and how address filtering will work. The frame structure drawing, then will show the address fields as part of the MRFI and not the network layer.

6.5 Frame details

The general frame layout is shown below. Detailed descriptions follow.

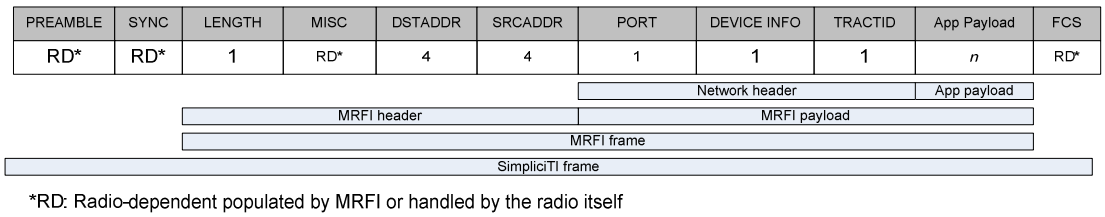


Figure 7: SimpliciTI frame structure (without Security enabled)

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

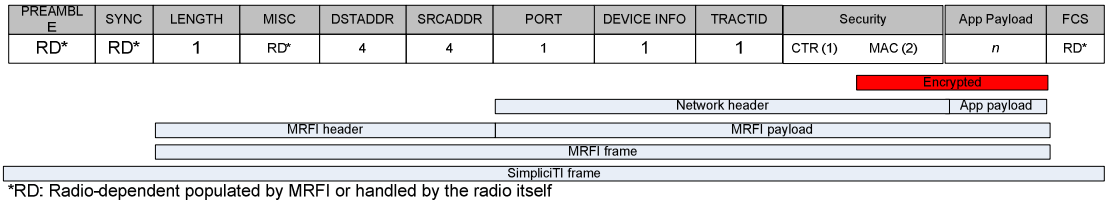


Figure 8: SimpliciTI frame structure (with Security enabled)

Field	Definition	Comments
PREAMBLE	Radio synchronization	Inserted by radio HW
SYNC	Radio synchronization	Inserted by radio HW.
LENGTH	Length of remaining packet in bytes	Inserted by FW on Tx. Partially filterable on Rx.
MISC	Miscellaneous frame fields	Differ for different radios. May be absent.
DSTADDR	Destination address	Inserted by FW. Filterable depending on radio.
SRCADDR	Source address	Inserted by FW.
PORT	Forwarded frame (7), Encryption context (6) Application port number (5-0).	Inserted by FW. Port namespace reserves 0x20-0x3F for customer applications and 0-1F for **NWK** management.
DEVICE INFO	Sender/receiver and platform capabilities	Inserted by FW. Details below.
TRACTID	Transaction id	Inserted by FW. Discipline depends on context. Need not be sequential.
APP PAYLOAD	Application data	$0 \leq n \leq 50$ for non-802.15.4 radios; $0 \leq n \leq 111$ for 802.15.4 radios
FCS	Frame Check Sequence	Usually a CRC appended by the radio hardware.

Table 1: SimpliciTI frame field summary

Some details are discussed below.

6.5.1 PREAMBLE/SYNC/LENGTH

The preamble and sync fields are inserted by the radio hardware.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

If such an option exists the radio will be configured to send and receive variable length frames[1]. When so configured a length byte is expected in the frame. This is the byte after the last sync byte is expected to be the length of the frame (not including the length byte itself).

6.5.2 MISC

This field may be absent. Some radios, e.g., IEEE 802.15.4 radios, have information such as frame control bytes in this field.

6.5.3 DSTADDR/SRCADDR

The destination and source addresses are treated as byte arrays. MRFI handles mapping the address to the filtering capabilities of the radio depending on whether the network layer requests filtering . To the extent that hardware filtering is not available across the address array software filtering is implemented by MRFI when filtering is active.

6.5.4 Security Context/PORT

Bits 7-6 hold the security context as defined in the table below:

Bit	Description	Comments
7	Forwarded frame	This bit is set when an Access Point forwards a frame to a polling device on behalf of the sender.
6	0: No encryption 1: Encryption enabled	Default is no encryption.
5-0	Port number	Abstraction denoting the application for which the frame is destined.

Table 2: Security context and Port number

Ports are conceptual abstractions that specify the target application handling the frame. The Port numbers **`0x00–0x1F`** are reserved or assigned as "well known Ports:" with specific services. They are intended for use by network management.

Port numbers **`0x20–0x3F`** are mapped to user handlers. Port **`0x3F`** is reserved as a broadcast Port to be used when the sending device has not been explicitly linked[2] to the receiving device. Port numbers starting at **`0x3E`** and decreasing are reserved for static allocation by user configurations. This arrangement is intended to accommodate commissioning scenarios.

The Ports are mapped by the network layer to Link IDs which are handles provided to the customer application when a link or link-listen succeeds. The application then uses this handle when messages are sent or received.

A conceptual representation of the port abstraction is shown in the following figure. Not all the well known ports are represented.

[1] Frames must be ≤ 64 bytes so that no frame will be larger than the transmit or receive FIFO that is minimal for the radios supported.

[2] Explicit linking *could* be done at build time as well as run time depending on the Customer's needs.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

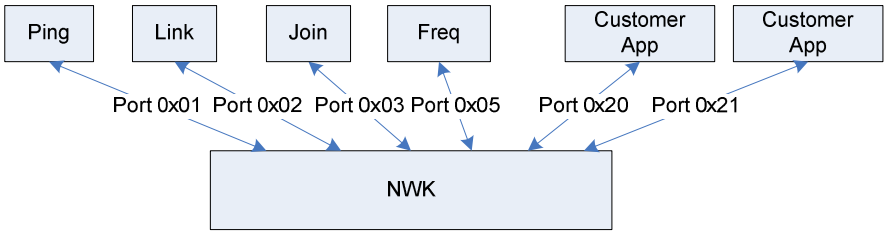


Figure 9: SimpliciTI port abstraction

6.5.5 DEVICE INFO

Device information byte provides information about the device issuing the frame.

The device information bits are defined as follows:

Bit	Description	Comments
7	Acknowledgement request	This bit is set by the **NWK** layer when a user application requests and acknowledgment from the peer.
6	00: Controlled listen 01: Sleeps/polls	Receiver type Sleeping devices may either poll or periodically listen. If they are specified as polling devices the Access Point will provide store-and-forward support. Otherwise the device listens at its own schedule under its own control.
5-4	00: End Device 01: Range Extender 10: Access Point 11: Reserved	Sender Type Most important for Range Extender to prevent a RE from forwarding a frame from another RE. This mitigates broadcast storms. Refers to device sending the frame and may not be the same as the device whose source address is specified in the frame.
3	Acknowledgement reply	This bit is set when the **NWK** layer sends a frame in response to an acknowledgment request.
2-0	Hop count	Decremented by each transmitting device until 0. Then discarded.

Table 3: DEVICE INFO bit values

6.5.6 TRACTID

The transaction ID is used by **NWK** to add robustness to the communications. The field can be used for example to match replies to outstanding messages or to help recognize a duplicate frame. Each sending side maintains its own transaction ID discipline.

7. NWK applications

The network layer applications on well-known ports are peer-to-peer objects used to manage the network. Some applications are mandatory. Some are conditional in that they must be present under certain conditions. A list follows. The conditions under which the conditional applications need be supported are noted.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Application	Port	Device Support
Ping	0x01	Mandatory except for Tx-only devices.
Link	0x02	Mandatory
Join	0x03	Only functional when the network has AP support.
Security	0x04	Only functional when the network has AP support.
Freq	0x05	Only functional when the network has AP support.
Mgmt	0x06	General use **NWK** management application. Used as the poll port.

Table 4: NWK applications

The first byte in the payload for each **NWK** application is an information byte. This byte conveys application-specific information. For example, there could be length information or option information such as whether to acknowledge the frame. In addition each application has its own payload discipline. Length information will be almost mandatory if block cipher encryption is used because in this case the payload length cannot be inferred from the frame length.

The payloads are described below.

7.1 Ping (Port 0x01)

7.1.1 Semantics

The purpose of this application is to detect the presence of a specific device.

This application sends a message from one device to another device. The receiving device echoes the payload contents back to the originator. The sequence is blocking in the sense that the originator will wait for the reply. A timeout parameter is applied.

Pings are unicast.

7.1.2 Payload (Client)

Address: unicast

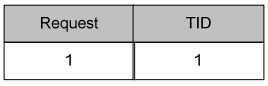


Figure 10: Ping Client payload

The payload consists of just the transaction ID (TID). The TID (transaction ID) is used to detect missing or duplicate frames.

7.1.3 Payload (Server)

Address: unicast

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

REQ Reply	TID
1(0x8x)	1

Figure 11: Ping Server payload

The payload consists of echoing the transaction ID (TID). The TID (transaction ID) is used to detect missing or duplicate frames. The msb in the application information byte signifies that this is a reply.

7.2 Link (Port 0x02)

7.2.1 Semantics

The purpose of this application is to support the link management of two peers.

Currently supported requests on the Link port are as follows:

Request	Value	Comments
Link	1	Sender requests a link (connection) be established.
Unlink	2	Tear down link (connection).

Table 5: Link request values

7.2.2 Link request

The frame source device (Client) is linked to the destination device (Server).[3] Once the link is established each side can send messages to the other. The connection is bi-directional.

The Client's link message is broadcast. The Server's reply is unicast back to the client. The Server remembers the Client's address from the Client's broadcast message. The Client remembers the Server's address from the unicast reply.

The complete exchange is necessary for the link to be established. At the completion of this function a bi-directional connection may exist between the two devices.

From the Client side the sequence is blocking in the sense that the Client will wait for the reply. A timeout parameter is applied. From the Server side listen call is also blocking. also with a timeout parameter.

In this simple protocol only one link per listen will be permitted, first come first served. Presumably the Client and Server will be engaged in the Link exchange based on some user intervention such as a button press, though this isn't required.

Multiple devices can be linked by doing multiple Link sessions. Each Client will be provided with a unique Access Token. A Client can also talk to multiple Servers again by instigating separate Link sessions.

7.2.2.1 Client side

Address: broadcast

[3] The Client-Server peer locus is arbitrary. The link transaction simply requires one talker to send a link frame and one listener to hear it. These are called Client and Server respectively here to distinguish the roles during the transaction. The actual application may or may not maintain these roles.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Request	TID	Link Token	Local Port	Rx Type	Protocol Ver
1 (0x01)	1	4	1	1	1

Figure 12: Client side Link payload (no Security)

Request	TID	Link Token	Local Port	Rx Type	Protocol Ver	CTR Value
1 (0x01)	1	4	1	1	1	4

Figure 13: Client side Link payload (with Security)

The payload consists of a request byte, a transaction ID, the 4-byte link token, a local port number, the Rx type of the Client (see Section 6.5.5), and the protocol version of the source.

The transaction ID (TID) is used by the link listening application to filter out duplicates.

The token is used to uniquely identify the link context and should be used to prevent rogue devices (either intended or unintended) from linking to the network. The customer can set these tokens to be unique in any manner. It is the means by which a device is prevented from linking to a device on the incorrect network. The token is a network-wide parameter. See Section 10.1.1.

The local port number is the port number that the remote device should use to communicate back to the linking device. If the linking device is Tx-only or does not need to receive any messages over this connection then a value of 0 will be supplied. In this case the connection is unidirectional.

The Rx type is used by the listener as a hint about setting the hop count when sending a frame to the client. If the peer is a polling device then sending to the Client can use a hop count that is large enough to get only to the Access Point. Otherwise, the hop count can be set to the number of hops it took to get to the listener which can be deduced from the received frame[4].

The protocol version can be used to determine run-time context. This can include denying the link if the versions do not match or compatibility modes if older versions are to be respected.

If security is enabled a modified **CTR** mode security is used. In this case the frame contains a four-byte counter value with which the sending device will begin when messages are sent across the connection.

In topologies with an AP the AP will supply the link token to joining devices. In topologies without APs the default link token is used.

7.2.2.2 Server side

Address: unicast

REQ Reply	TID	Local Port	Rx Type
1 (0x81)	1	1	1

Figure 14: Server side Link response payload (no Security)

REQ Reply	TID	Local Port	Rx Type	CTR Value
1 (0x81)	1	1	1	4

Figure 15: Server side Link response payload (with Security)

[4] This may not be a safe practice if the peer such as a remote control device can move.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

The response payload contains a transaction ID, a local port number and the Server side Rx type.

The Local Port object will be used to populate the PORT field in the frame when the Client's linked application sends a message to its peer on the Server. The association is maintained in the **NWK** layer.

The Rx type is used for the same purpose as the Client side, i.e. as a hint to how to set the hop count in frames sent to the Server.

As with the Client side, if security is enabled a modified **CTR** mode security is used. In this case the reply frame contains a four-byte counter value with which this sending device will begin when messages are sent across the connection. Note that messages may not be sent in both directions though bi-directional messaging is supported.

The most significant bit in the REQ Reply byte signifies that this is a reply.

7.2.3 Unlink request

Either peer may request that a connection be terminated. The requesting side waits for the results of the request from the peer. The peer will reply if possible. This request is supplied for completeness. A connection-based protocol like SimpliciTI could lead to awkward designs if the connection base is too dynamic.

The unlink request originator receives a result code indicating the status of the peer that is the target of the request. If the peer does not respond[5] the application is notified. Normally the originator will reclaim the connection resources on a successful result. Other corrective action is possible if the unlink fails with respect to the peer.

The peer also will normally reclaim connection resources. Since it is possible that the peer will miss the originator's request application designs should account for this.

7.2.3.1 Originator side

Address: Unicast

Request	TID	Remote port
1 (0x02)	1	1

Figure 16: Originator side Unlink payload

The originator supplies the port on which it receives messages from the peer. This is the remote port from the peer's perspective. The peer uses this value together with the source address in the frame to find the correct entry in its Connection table.

7.2.3.2 Peer side

Address: Unicast

REQ Reply	TID	Result code
1 (0x82)	1	1

Figure 17: Peer side Unlink reply payload

The result code indicates whether the peer successfully destroyed its side of the connection.

[5] For example the peer may not be listening when the request is sent.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

7.3 Join (Port 0x03)

7.3.1 Semantics

This application guards entry to the network in topologies with APs.

When the network has an AP each device must join the network. Upon joining a device is optionally supplied with the following:

- Encryption context
- Link token
- Store-and-forward support if it is a sleeping Rx device

By supplying the first three items the network is more robust to intrusion by rogue devices.

The attempt to join should timeout in case the AP isn't up yet. It is up to the Customer application to keep trying to join. Since the presence of an AP is a build-time parameter all devices know how to behave.

Currently supported join request bytes conveys are as follows:

Request	Value	Comments
Join	1	Sender requests to Join.

Table 6: Join request values

7.3.1.1 Join request: Client side

Address: broadcast

Request	TID	Join Token	Number of connections	Protocol Ver
1 (0x01)	1	4	1	1

Figure 18: Join Client side payload

The payload consists of a request byte, a transaction ID, the 4-byte join token, the number of connections allowed on the source, and the protocol version of the source.

As in previous descriptions the transaction ID (TID) is used by the application to filter out duplicates.

The Join Token is used to verify that the device is valid on the network.

The number of connections is used by the Access Point (if present) as a hint to allocate resources. In the topology in which the Access Point functions as a data hub in a Star configuration the receipt of a Join frame can be used to stimulate a listening for the joining client to link. If the joining client does not intend to link (such as a Range Extender device) the number of connections field should be set to 0. The Access Point can then infer that there will be no peers on the joining device so it need not listen for a link frame or allocate connection table resources.

The protocol version can be used to determine run-time context. This can include denying the join if the versions do not match or compatibility modes if older versions are to be respected.

7.3.1.2 Join request: Server side

Address: unicast

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

REQ Reply	TID	Link Token	FUNC/LEN	KEY
1 (0x81)	1	4	1	n

Figure 19: Join Server side payload

The Server returns the source TID and the value that the Client should use as the link token for the network. Encryption context is also returned. The next bytes use the same format as those in the Security application (see Section 7.4.1.1).

The msb in the application information byte signifies that this is a reply.

7.4 Security (Port 0x04)

7.4.1 Semantics

This application is used to change security information such as encryption keys and encryption context. For this application the AP initiates the exchange so the application appears only in networks that have an AP and then only on non Tx-only End Devices and Range Extenders.

Having the AP initiate these exchanges also helps with sleeping Rx devices that are managed by the AP so that they do not get out of synchronization. The AP can maintain multiple encryption contexts and alert the sleeping devices when they awaken and poll.

The AP uses this application to enforce the encryption context and communicate a new key to all other devices.

These frames will always be sent encrypted using the default encryption key.

7.4.1.1 Initiator (AP) side

Address: broadcast

Request	TID	FUNC/LEN	KEY
1	1	1	n

Figure 20: Security initiator (AP) payload

As in previous descriptions the transaction ID (TID) is used by the application to filter out duplicates. The next bytes are used as follows:

Field	Definition	Description
FUNC/LEN	N/A	Reserved – currently unused.

Table 7: Security initiator payload details

7.5 Freq (Port 0x05)

7.5.1 Semantics

The protocol on this port supports the following operations. Details follow.

Request	Value	Comments
Change Channel	1	Only the AP can issue this command.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Echo request	2	This is functionally a frequency application ping that is used by a device to scan for the current channel. Only the AP will respond to this frame
Change channel request	3	A non-AP device can request a channel change. But the command must still be issued by the AP. Future support.

When the network has an AP this application provides frequency agility support. The AP initiates the change in frequency by sending a broadcast frame to the Freq Port announcing the frequency change context.

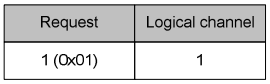
If the frequency of the network changes sleeping devices will find the AP by cycling through the frequency table upon awakening. They will know to do so because they will not receive an **ACK** when they poll. The frequency application echo request is used for this scan.

There is provision for a non-AP device to request a change of channels. The actual change is initiated by the AP at its discretion.

7.5.1.1 Change channel command

7.5.1.1.1 Initiator (AP) side

Address: broadcast



Request	Logical channel
1 (0x01)	1

Figure 21: FREQ application change channel command

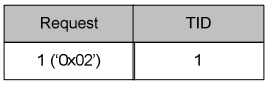
The payload consists of the request indicator '1' and the logical channel number of the new radio channel. The target channel is a logical channel number that is most directly implemented as an index into a table containing everything required for a frequency change context.

A transaction ID is not needed for this frame. It is broadcast so replies will not occur and therefore do not need to be matched with sent frames. On the receiver side it does no real harm if duplicates are received. This is unlikely anyway since the receipt of any frame will cause the channel to change.

7.5.1.2 Echo request (Frequency Application ping)

7.5.1.2.1 Client side

Address: broadcast/unicast to AP



Request	TID
1 ('0x02')	1

Figure 22: FREQ application echo request (ping)

The payload consists of the request indicator '2' and the transaction ID. The client side sends this frame to the Access Point if it has gleaned the AP address from the Join reply. Otherwise it broadcasts the frame. A broadcast is necessary if the scan is done at startup before the Join has succeeded.

7.5.1.2.2 Server side

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Address: unicast

REQ Reply	TID
1 ('0x82')	1

Figure 23: FREQ application echo reply

The payload consists of the reply indicator '0x82' (original command value with msb turned on).and the TID.

The client side tries the ping on each channel until this reply is received. The client then can infer the current channel being supported.

7.5.1.3 Channel change request

Using this frame a non-AP device can request that the Access Point issue a change channel command. The AP need not comply.

There is no reply to this command. If the AP decides to comply it will issue a channel-change command (see Section 7.5.1.1.1).

7.5.1.3.1 Initiator side

Address: unicast to Access Point

Request
1 ('0x03')

Figure 24: FREQ application change channel request

The payload consists of the command indicator '3'. There will be no reply to this frame and no transaction ID is needed.

7.6 Mgmt (Port 0x06)

7.6.1 Semantics

This is a general management port to be used to manage the device. It will be used to access a device out-of-band and could be used, for example, to reset other Port state machines, Transaction IDs, etc. It will also be used by sleeping devices to poll the AP.

This is the port used by sleeping/polling end devices to poll the AP for frames.

This can be used as an emergency meet-me port. Messages can be sent in clear text to this port at any time.

7.6.1.1 End device polling

An End Device announces its intent to poll the Access Point for messages when it joins the network. This context is carried in the **DEVICE INFO** byte (see 6.5.5). The poll query is sent on the **Mgmt** port. The server-side reply is sent on the port about which the query was made. Note that this query is sent as a side effect of a receive call and is not part of the SimpliciTI API.

7.6.1.1.1 Client side

Address: unicast to Access Point

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Request	TID	Port	Address
1	1	1	4

Figure 25: End Device Poll

The query of the Access Point is made for a specific port and address. The port is derived from the Link ID supplied in the receive call originating in the application. The address is the address of the client's peer, i.e., the SimpliciTI address of the peer from which the client is expecting a frame.

The transaction ID is used by the AP so that it can discern duplicate poll frames and not send a frame prematurely to the client.

7.6.1.1.2 Server side

The reply from the Server (Access Point) is not on the **Mgmt** port. The Server replies on the port queried.

If there is a frame waiting for the client on the port queried the Server sends the frame. If no frames are waiting on that port the Server constructs a frame with no payload and sends it to that port on the Client. Both these frames use the transaction ID sup[plied in the poll request.

8. API

The focus of the API is to encapsulate network functionality in a manner that allows a functional reliable network to form with little effort from the application. The main effect of this approach is that the resulting network sacrifices flexibility for simplicity.

The following functionality is supported by the API:

- Initialization
- Linking
- Application peer-to-peer messaging
- Device management

At the **NWK** layer most but not all exchanges have acknowledgments. At the application layer the acknowledgment status is controlled by the application peer-to-peer

8.1 smplStatus_t SMPL_Init(uint8 (*pCB)(linkID))

This call causes all **NWK** initialization to occur. A side effect of the initialization is an attempt to Join the network. If it is an Access Point supported network the Join request (issued silently) will negotiate network parameters in exchanges with the Access Point.

This call also causes the radio to be set up and initializes all **NWK** constructs based on system parameters defined in Section 10. If the device is an AP this may involve such other actions as generating an encryption key and a link token.

The argument is a pointer to a function that takes a Link ID argument and returns a uint8. This allows the application to provide callback function pointer to the frame handling logic for received frames. If the following conditions are met the callback will be invoked with the Link ID of the received frame as the argument

1. The supplied function pointer is not null
2. The destination port of the received frame is a user port
3. The destination address of the frame is that of the device
4. The connection is valid (i.e., the port is in the connection table).

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

If the function returns non-zero the frame is considered as having been handled and the frame resources are released for re-use. Otherwise the frame remains in the input frame queue for later handling by the application. In either case the application retrieves the received frame by executing a call to **`SMPL_Receive()`** with the link ID conveyed in the callback argument as the link ID argument. This call is guaranteed to succeed.

If the device is an Access Point and the AP is designated as a data hub the callback will also be invoked if the joining device supports End Device objects. In this case the Link ID will have the value 0x00. In this case the application cannot retrieve the received frame directly since it is handled in the **`NWK`** layer. The callback is used as a signal to the AP that it should execute a **`SMPL_LinkListen()`** to receive the subsequent link frame from the joining device.

Note that if the callback conditions are met the handler will be invoked in the receive ISR context. Designers should be careful about doing too much in this thread.

8.2 smplStatus_t SMPL _Link(linkID_t *linkID)

Link as a Client to another device that is listening (Server).

Listening devices will acknowledge. Once the acknowledgment as been received the call will return with a link ID that should be used for all subsequent messaging to and from the device to which the link has been made. This is not a blocking call and will return status that indicates whether the link succeeded. If it did not succeed the application can re-try.

8.3 smplStatus_t SMPL _Unlink(linkID_t linkID)

Tear down the connection bound to the supplied Link ID. Part of Extended API.

This call will unconditionally disable the local connection table entry bound to the supplied Link ID. In addition a message will be sent to the Link Port of the peer requesting that the peer tear down its side of the connection (see Section 7.2.3). The disabling of the connection on the peer may or may not succeed when initiated from the local device.

8.4 smplStatus_t SMPL _LinkListen(linkID_t *linkID)

This is the companion to the Link call and waits for a Client Link message.

Only the first valid Link message is accepted as a result of this call. Only one outstanding listen can be in force at any given time. Link messages that arrive before the listen is executed are discarded.

There is some defense against receiving duplicate link frames during a listen. The defense is designed to detect retransmissions or late frames. But it will also detect link frames coming from a device that was reset provided that the reset device generates the same link frame each time. For example, a device that randomly generates a new device address each time will not be detected as a duplicate linking device and will consume resources.

This is a timed blocking call and will return when either a valid link frame is received or a (configurable) fixed amount of time has elapsed. The application can discriminate between these two by the return code. The application is then free to implement a recovery policy including another **SMPL_LinkListen()**.

8.5 smplStatus_t SMPL _SendOpt(linkID_t lid, uint8 *msg, uint8 len, uint8 opt)

This call sends the message **msg** of length **len** to the device with link id **lid** applying the send options present in the **opt** bit map. The **lid** is the one returned by the **SMPL_Link()** call.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

It returns after the message is sent. This is so that discipline can be maintained over the transmit context and the power context of the MCU and the radio.

The message size is limited to the transmit FIFO size which depends on the radio. Segmentation and reassembly if needed is the responsibility of the application

8.6 smplStatus_t SMPL _Send(linkID_t lid, uint8 *msg, uint8 len)

This API is maintained for legacy reasons. Initial version of SimpliciTI did not support the transmit option argument.

8.7 smplStatus_t SMPL _Receive(linkID_t lid, uint8 *msg, uint8 *len)

This call checks the frame buffers for a message from link id **lid** and returns the oldest one and its length. The frame buffers are populated as a result of handling a radio Rx interrupt. The **SMPL_Receive()** call does **not** affect the radio state, i.e., it does **not** turn the radio on or place it into the Rx state. It simply checks to see if a relevant frame has been received in an ISR thread and is being held in the frame buffer.

The Link ID supplied in the call is one originally defined by having completed a successful link operation.

8.8 smplStatus_t SMPL_Ping(linkID_t lid)

This API is part of the Extended API set. It provides application level access to the **NWK** ping application. It is a simple means by which an application can determine the presence of the device on which a peer resides.

This call does *not* ping the peer itself. The Link ID supplied in the formal parameter is sued to glean the address of the device hosting the peer. The API causes the NWK to send a ping frame to the Pi9ng NWK application on the device hosting the peer. The result of this trasnction is returned to the caller.

There is a useful side effect of this call if Frequency Agility is enabled. The call will result in a channel scan if a reply is not received on the current channel. A return code indicating success implies both that the peer device is there and that the channel setting on the local device is current.

8.9 smplStatus_t SMPL_Commission (addr_t *addr, uint8_t lPort, uint8_t rPort, linkID_t *lid);

This API is part of the Extended API set. It is used to do static configuration of a peer connection. It can be used to eliminate the need for a link session between peers. It is a hard-wired connection configuration that requires knowledge of the peer device address (**addr**) and both the local (**lPort**) and remote (**rPort**) port assignments. On successful return from this call a Link ID (**lid**) is provided. This value is assigned by the **NWK**.

The local port namespace reserves a defined area for static assignment. The size of this area is a build time setting. The default size is 1. The automatic port assignment mechanism will not use this part of the namespace when used for explicit link sessions.

8.10 void SMPL_Ioctl(ioctlObject_t object, ioctlAction_t action, void *val)

This is the means by which the application can configure the **NWK** at run time. The currently available values are described in Section 10.2.

8.11 Pseudo-code example

The following pseudo-code snippet shows the API sequence for how a device would join a SimpliciTI network with an AP and then link to two other devices and send different messages.

```
void main()
{
```

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

```
  linkID_t linkIDLow, linkIDHigh;
  uint32   temp;

  // Initialize the board's HW
  BSP_InitBoard();

  // Initialize SimpliciTI. The initialization will cause the
  // device to Join as a side effect. The Join will return
  // security key and a token to be used in linking
  // sequences for this network. This is all hidden from
  // the application.

  SMPL_Init(0);  // no callback supplied

  // Establish links to two different (logical) devices.
  // One will get a message if the sampled temperature is
  // to low. The other gets a message if it is too high.
  SMPL _Link(&linkIDLow);
  SMPL _Link(&linkIDHigh);
  while (TRUE)
  {
    // put board to sleep until timer wakes it up
    // to read the temperature sensor
    MCU_Sleep();
    HW_ReadTempSensor(&temp);
    if (temp > TOO_HIGH)
    {
      SMPL _Send(linkIDHigh, "Hot!", 4);
    }
    if (temp < TOO_LOW)
    {
      SMPL _Send(linkIDLow, "Cold!", 5);
    }
  }
}
```

When this device is linking the target device must have executed a **`SMPL_LinkListen()`** first. This is how the linked pairs are identified. The two links need not be on the same physical device.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

There would be a **SMPL_Receive(linkID…)** after each of the **SMPL_Send()** statements if the application has implemented acknowledgments. The receive activity might have to be conditioned with a timeout discipline to know whether to re-transmit the message if that were the intent. More likely, though, this example would be simple unacknowledged messages unless missing one was critical.

9. Sequence diagrams and state machines

10. Customer configurable objects

10.1 Build time

10.1.1 Non-radio items

The following are the build-time configurable items. Each has a default value so none actually need to be modified by the customer. Only those currently supported are listed. The macros defining the values are found in the files **smpl_config.dat** and **smpl_nwk_config.dat**, a pair for each of the 3 device types.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Item	Default value	Description
MAX_HOPS	3	Maximum number of times a frame is resent before frame dropped. Each RE and the AP decrement the hop count and resend the frame.
MAX_HOPS_FROM_AP	1	Maximum distance and ED can be from the AP. Using this hop count significantly reduces the broadcast storm that can result from an ED poll if **MAX_HOPS** is used. Only for AP networks.
NUM_CONNECTIONS	4	Number of links supported as a result of both **SMPL_Link()** and **SMPL_LinkListen()** calls. Should be 0 if the device supports no ED objects (APs or REs).
MAX_APP_PAYLOAD	10	Maximum number of bytes in the application payload.
SIZE_INRAME_Q	2	Number of frames held in the Rx frame queue. Can be 0 for Tx-only devices, or for devices that never receive frames.
SIZE_OUTFRAME_Q	2	Number of frames held in the Tx frame queue. Some **NWK** applications keep Tx frames around to match replies.
DEFAULT_JOIN_TOKEN	**0x01020304**	Joining a network requires this value to match on all devices. It is sent in the Join message and matched in the receiving Access Point.
DEFAULT_LINK_TOKEN	**0x05060708**	Obtaining link access to a network device requires this value to match on all devices. It is sent in the Link message and matched in the receiving device.
THIS_DEVICE_ADDRESS	**0x12345678**	Each device address must be unique. The address assignment should lock out **0xnnnnnn00** and **0xnnnnnnFF** which are broadcast address for the CC1100/CC2500-class radios.
FREQUENCY_AGILITY	**Not defined**	When defined enabled support for Frequency Agility. Otherwise only the first entry in the channel table is used.
NVOBJECT_SUPPORT	**Not defined**	Support for saving and restoring connection context.
SMPL_SECURE	**Not defined**	Enables SimpliciTI security support.
APP_AUTO_ACK	**Not defined**	Support for application layer acknowledgment support.
EXTENDED_API	**Not defined**	Support for `SMPL_Ping()`, `SMPL_Unlink()`, and `SMPL_Commission()`.
SW_TIMER	**Not defined**	If enabled uses software to implement delays.
Access Point Devices		
ACCESS_POINT	**Defined**	
NUM_STORE_AND_FWD_CLIENTS	**10**	Number polling End Devices supported by this Access Point
AP_IS_DATA_HUB	**Not defined**	If this macro is defined the AP will be notified through the callback each time a device joins. The AP should be running an application that listens for a link message on receipt of this notification. The ED joining must link immediately after it receives the Join reply.
Range Extender Devices		
RANGE_EXTENDER	**Defined**	
End Devices		
END_DEVICE	**Defined**	Defined unless it is a application hosted on a device tht is

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

		also either a Range Extender or Access Point.
RX_POLLS	**Not defined**	Define if the device polls for frames.

Table 8: Build-time Customer configurable non-radio parameters

10.1.2 Radio configuration

A large number of radio parameters are available to be read and modified if necessary. The SimpliciTI environment will be shipped with default settings for these parameters.

Basically they are the parameters set by SmartRF. They will not be listed individually here. Some may not have the default SmartRF values and some may be subsequently modified by startup code either by **NWK** or application either directly using **SMPL_Ioctl()** or indirectly during startup exchanges.

With few exceptions these parameters will not be directly accessible to customers through the SimpliciTI API. If a customer needs to modify any of these parameters they will have access to them directly in the code but they will not be supported with an API.

10.2 Run time

Run time configuration of **NWK** objects is accomplished using the **SMPL_Ioctl(*object, request, value*)** API. The following Table summarizes the run-time objects.

Object	Description	Comments
IOCTL_OBJ_FREQ	Get/Set radio frequency	Frequency agility. May be used by application or **NWK**.
IOCTL_OBJ_CRYPTKEY	Set encryption key	Customer may provide external means for user to set a non-default key. Requires reset to take effect.
IOCTL_OBJ_RAW_IO	Application layer access to the frame header to directly send or receive a frame.	This object is used for example to ping another device where the network address of the target device is supplied directly and not done through the connection table.
IOCTL_OBJ_RADIO	Application layer access to some radio controls.	Limited access to radio directly. For example, sleeping and awakening the radio and getting signal strength information.
IOCTL_OBJ_AP_JOIN	Access Point join-allow context	Interface to control whether Access Point will allow devices to join or not.
IOCTL_OBJ_ADDR	Get/set device address	Permits run-time address generation for a device. Set function *must* be done before the **SMPL_Init()** call.
IOCTL_OBJ_CONNOBJ	Connection object	Delete a connection entry. Accessed with Link ID. Affects this device only – does not gracefully tear down connection.
IOCTL_OBJ_FWVER **IOCTL_OBJ_PROTOVER**	Firmware and protocol versions.	Get only. FW version a byte array of length **SMPL_FWVERSION_SIZE.** Protocol version a `uint8_t`.
IOCTL_OBJ_NVOBJ	Non-volatile memory object	Get and set the NV object. This permits a device to save connection context across power cycles so that communications can be restored in event of a reset.
IOCTL_OBJ_TOKEN	Link or Join token object	Allows application to get/set a network access token.

Table 9: Customer configurable run-time objects

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

B.2. Interfaz de Programación de Aplicaciones

SimpliciTI
Application Programming Interface

Document Number: SWRA221

Texas Instruments, Inc.
San Diego, California USA

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

Version	Description	Date
1.0	Initial release	08/01/2008
1.1	Update for the 1.1.0 release	01/14/2009
1.2	Updated title page	03/24/2009

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

TABLE OF CONTENTS

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

1. Introduction

1.1 Purpose

This document describes the application programming interface for SimpliciTI software. The API provides an interface to the services of the SimpliciTI protocol stack.

1.2 References

1. *SimpliciTI Specification Version 1.1.0.*
2. *SimpliciTI Developers Notes*

1.3 Font usage

There are a few special usage fonts:

Font	Usage
`Fixed pitch`	Used for file names, code snippets, symbols, and code examples.
Underlined blue normal text	Document cross reference hyperlink

1.4 Acronyms and Definitions

API	Application Programming Interface.
BSP	Board Support Package
CCA	Clear Channel Assessment
GPIO	General Purpose Input Output
ISR	Interrupt Service Routine
LED	Light Emitting Diode
LQI	Link Quality Indication.
LRU	Least Recently Used
MAC	Medium Access Control.
PHY	Physical layer.
RSSI	Received Signal Strength Indicator

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

2. API Overview

2.1 Interface Mechanisms

The following interface mechanisms are used in the SimpliciTI API.

2.1.1 Direct Execute Function Calls

These API functions directly execute code that performs an operation. The function executes in the context of the caller. These functions may have critical sections.

2.1.2 Callback Function

There is one optional callback opportunity in SimpliciTI. The function must be defined and implemented by the application and is registered during initialization. The callback function implementation should avoid CPU intensive operations as it runs in the ISR context. This function is described in detail in Section 7.

2.2 Data Interfaces

These interfaces support sending and receiving data between the SimpliciTI stack and the application and ultimately support the peer-to-peer messaging.

2.3 Common Constants and Structures

2.3.1 Common Data Types

The following are defined:

```
typedef   signed char     int8_t;
typedef   signed short    int16_t;
typedef   signed long     int32_t;
typedef   unsigned char   uint8_t;
typedef   unsigned short  uint16_t;
typedef   unsigned long   uint32_t;
typedef   unsigned char   linkID_t;
typedef   enum smplStatus smplStatus_t;
```

In addition a further set of types and structures are used for the `ioctl` interface. These are described in Section 6.

2.3.2 Status

The following status values are used in various API functions. They are of type `smplStatus_t`. The relevant return codes will be specified individually for each API symbol in the following sections.

NAME	DESCRIPTION
SMPL_SUCCESS	Operation successful.
SMPL_TIMEOUT	A synchronous invocation timed out.
SMPL_BAD_PARAM	Bad parameter value in call.
SMPL_NOMEM	No memory available. Object depends on API.
SMPL_NO_FRAME	No frame available in input frame queue.
SMPL_NO_LINK	No reply received to Link frame sent.
SMPL_NO_JOIN	No reply received to Join frame sent.
SMPL_NO_CHANNEL	Channel scan did not result in response on at least 1 channel.

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

SMPL_NO_PEER_UNLINK	Peer could not delete connection. Returned in reply message to unlink request. (Not officially supported yet.)
SMPL_TX_CCA_FAIL	Frame transmit failed because of CCA failure.
SMPL_NO_PAYLOAD	Frame received but with no application payload.
SMPL_NO_AP_ADDRESS	Should have previously gleaned an Access Point address but we have none.

2.3.3 Special Link IDs

SimpliciTI supports special Link IDs that are available to the application by default. The following values indicate the special Link IDs.

NAME	DESCRIPTION
SMPL_LINKID_USER_UUD	Unconnected User Datagram Link ID. This is a special, connectionless Link ID supported by default on all user applications.

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

3. Initialization Interface

3.1 Introduction

SimpliciTI initialization involves three stages of initialization: board, radio, and stack. Board initialization (BSP) is deliberately separated from the radio and stack initialization. The radio and stack initialization occur as a result of the SimpliciTI initialization call. The board initialization is a separate invocation not considered part of the SimpliciTI API but it is noted here for completeness.

The BSP initialization is partitioned out because customers may already have a BSP for their target devices. Making the BSP initialization explicit in the SimpliciTI distribution makes it easier to port to another target.

3.1.1 Board Initialization

SimpliciTI supports a minimal board-specific BSP. The BSP scope includes GPIO pin configuration for LEDs, switches, and a counter/timer used for protocol chores. It also includes SPI initialization for the dual-chip RF solutions.

3.1.2 Radio Initialization

Radio registers are populated and the radio is placed in the powered, idle state. Most of the radio registers are based on exported code from SmartRF Studio. The default channel is set with the first entry in the channel table.

3.1.3 Stack Initialization

All data structures and network applications are initialized. In addition the stack issues a Join request on behalf of the device. The Join request will fail in topologies in which there is no Access Point. This is expected in this topology and is not an error condition.

In topologies in which an Access Point is expected to be present Join failure is an error condition and the application should continue to retry or take other action.

3.2 BSP_Init ( )

3.2.1 Description

Not strictly part of the SimpliciTI API this call initializes the specific target hardware. It should be invoked before the `SMPL_Init()` call.

3.2.2 Prototype

```
void BSP_Init(void)
```

3.2.3 Parameter Details

None.

3.2.4 Return

None.

3.3 SMPL_Init ( )

3.3.1 Description

This function initializes the radio and the SimpliciTI protocol stack. It must be called once when the software system is started and before any other function in the SimpliciTI API is called.

3.3.2 Prototype

```
smplStatus_t SMPL__Init(uint8_t (*callback)(linkID_t))
```

3.3.3 Parameter Details

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

PARAMETER	DESCRIPTION
callback	Pointer to function that takes a linkID_t argument and returns a uint8_t.

A non-null argument causes the supplied function to be registered as the callback function for the device. Since the initialization is called only once the callback serves all logical End Devices on the platform.

The function is invoked in the frame-receive ISR thread so it runs in the interrupt context. Details of the callback are discussed in Section 7.

It is valid for this parameter to be null if no callback is supplied.

3.3.4 Return

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Initialization successful.
SMPL_NO_JOIN	No Join reply. Access Point possibly not yet up. Not an error if no Access Point in topology
SMPL_NO_CHANNEL	Only if Frequency Agility enabled. Channel scan failed. Access Point possibly not yet up.

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

4. Connection Interface

4.1 Introduction

This interface provides the mechanism to establish a connection between two peers.

4.2 SMPL_Link()

4.2.1 Description

This call sends a broadcast link frame and waits for a reply. Upon receiving a reply a connection is established between the two peers and a Link ID is assigned to be used by the application as a handle to the connection.

This call will wait for a reply but will return if it does not receive one within a timeout period so it is not a strictly blocking call. The amount of time it waits is scaled based on frame length and data rate and is automatically determined during initialization.

This call can be invoked multiple times to establish multiple logical connections. The peers may be on the same or different devices than previous connections.

4.2.2 Prototype

```
smplStatus_t SMPL_Link(linkID_t *lid)
```

4.2.3 Parameter Details

PARAMETER	DESCRIPTION
`lid`	The parameter is a pointer to a Link ID. If the call succeeds the value pointed to will be valid. It is then to be used in subsequent APIs to refer to the specific peer.

4.2.4 Return

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Link successful.
`SMPL_NO_LINK`	No Link reply received during wait window.
`SMPL_NOMEM`	No room to allocate local Rx port, no more room in Connection Table, or no room in output frame queue.
`SMPL_TX_CCA_FAIL`	Could not send Link frame.

4.3 SMPL_LinkListen()

4.3.1 Description

This call will listen for a broadcast Link frame. Upon receiving one it will send a reply directly to the sender.

This call is a modified blocking call. It will block “for a while” as described by the following constant set in the `nwk_api.c` source file:

CONSTANT	DESCRIPTION
`LINKLISTEN_MILLISECONDS_2_WAIT`	Number of milliseconds this thread should block to listen for a Link frame. The default is 5000 (5 seconds)

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

The application can implement a recovery strategy if the listen times out. This includes establishing another listen window. Note that there is a race condition in that if the listen call is invoked upon a timeout it is possible that a link frame arrives during the short time the listener is not listening.

4.3.2 Prototype

```
smplStatus_t SMPL_LinkListen(linkID_T *lid)
```

4.3.3 Parameter Details

PARAMETER	DESCRIPTION
`lid`	The parameter is a pointer to a Link ID. If the call succeeds the value pointed to will be valid. It is then to be used in subsequent APIs to refer to the specific peer.

4.3.4 Return

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Link successful.
`SMPL_TIMEOUT`	No link frame received during listen interval. Link ID not valid.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

5. Data Interface

5.1 Introduction

This API provides interfaces to send and receive data between peers.

5.2 SMPL_SendOpt ( )

5.2.1 Description

This function sends application data to a peer with the capability of specifying transmit options. The network code takes care of properly conditioning the radio for the transaction. Upon completion of this call the radio will be in the same state it was before the call was made. The application is under no obligation to condition the radio.

By default the transmit attempt always enforces CCA.

5.2.2 Prototype

```
smplStatus_t SMPL_SendOpt(linkID_t lid, uint8_t *msg, uint8_t len. txOpt_t opts)
```

5.2.3 Parameter Details

PARAMETER	DESCRIPTION
`lid`	Link ID of peer to which to send the message.
`msg`	Pointer to message buffer.
`len`	Length of message. This can be 0. It is legal to send a frame with no application payload.
`opts`	Bit map of valid options selected for the transmit

The '`lid`' parameter must be one established previously by a successful Link transaction. The exception is the Unconnected User Datagram Link ID (see Section 2.3.3). This Link ID is always valid. Since this Link ID is not connection-based a message using this Link ID is effectively a datagram sent to all applications.

Valid transmit options are:

Option (macro)	Description
`SMPL_TXOPTION_NONE`	No options selected.
`SMPL_TXOPTION_ACKREQ`	Request acknowledgement from peer. Synchronous call.

5.2.4 Return

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Transmission successful.
`SMPL_BAD_PARAM`	No valid Connection Table entry for Link ID; data in Connection Table entry bad; no message or message too long.
`SMPL_NOMEM`	No room in output frame queue.
`SMPL_TX_CCA_FAIL`	CCA failure. Message not sent.
`SMPL_NO_ACK`	No acknowledgment received.

5.3 SMPL_Send ( )

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

5.3.1 Description

This function sends application data to a peer. This API provides legacy support for SimpliciTI releases that predate the addition of the transmit options. This API is equivalent to calling `SMPL_SendOpt()` with `SMPL_TXOPTION_NONE` specified.

The network code takes care of properly conditioning the radio for the transaction. Upon completion of this call the radio will be in the same state it was before the call was made. The application is under no obligation to condition the radio.

By default the transmit attempt always enforces CCA.

5.3.2 Prototype

```
smplStatus_t SMPL_Send(linkID_t lid, uint8_t *msg, uint8_t len)
```

5.3.3 Parameter Details

PARAMETER	DESCRIPTION
`lid`	Link ID of peer to which to send the message.
`msg`	Pointer to message buffer.
`len`	Length of message. This can be 0. It is legal to send a frame with no application payload.

The '`lid`' parameter must be one established previously by a successful Link transaction. The exception is the Unconnected User Datagram Link ID (see Section 2.3.3). This Link ID is always valid. Since this Link ID is not connection-based a message using this Link ID is effectively a datagram sent to all applications.

5.3.4 Return

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Transmission successful.
`SMPL_BAD_PARAM`	No valid Connection Table entry for Link ID; data in Connection Table entry bad; no message or message too long.
`SMPL_NOMEM`	No room in output frame queue.
`SMPL_TX_CCA_FAIL`	CCA failure. Message not sent.

5.4 SMPL_Receive ( )

5.4.1 Description

This function checks the input frame queue for any frames received from a specific peer.

Unless the device is a polling device this call does **not** activate the radio or change the radio's state to receive. It only checks to see if a frame has already been received on the specified connection.

If the device is a polling device as specified in the device configuration file (see Section 9.2 in the Developers Notes) the network layer will take care of the radio state to enable the device to send the polling request and receive the reply. In this case conditioning the radio is not the responsibility of the application.

If more than one frame is available for the specified peer they are returned in first-in-first-out order. Thus it takes multiple calls to retrieve multiple frames.

5.4.2 Prototype

```
smplStatus_t SMPL_Receive(linkID_t lid, uint8_t *msg, uint8_t *len)
```

5.4.3 Parameter Details

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

PARAMETER	DESCRIPTION
lid	Check for messages from the peer specified by this Link ID.
msg	Pointer to message buffer to populate with received message.
len	Pointer to location in which to save length of received message.

The 'lid' parameter must be one established previously by a successful Link transaction. The exception is the Unconnected User Datagram Link ID (see Section 2.3.3). This Link ID is always valid. The application must ensure that the message buffer is large enough to receive the message. To avoid a buffer overrun the best strategy is to supply a buffer that is as large as the maximum application payload specified in the network configuration file (MAX_APP_PAYLOAD) used during the project build.

5.4.4 Return

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Frame for the Link ID found. Contents of 'msg' and 'len ' are valid.
SMPL_BAD_PARAM	No valid Connection Table entry for Link ID; data in Connection Table entry bad.
SMPL_NO_FRAME	No frame available.
SMPL_NO_PAYLOAD	Frame received with no payload. Not necessarily an error and could be deduced by application because the returned length will be 0.
SMPL_TIMEOUT	Polling Device: No reply from Access Point.
SMPL_NO_AP_ADDRESS	Polling Device: Access Point address not known.
SMPL_TX_CCA_FAIL	Polling Device: Could not send data request to Access Point
SMPL_NOMEM	Polling Device: No memory in output frame queue
SMPL_NO_CHANNEL	Polling Device: Frequency Agility enabled and could not find channel.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

6. Device Management: IOCTL Interface

6.1 Introduction

The `ioctl`[1] interface is the means by which applications can get access to more refined control over the device. There is a general form for the interface that specifies an object, and action, and any parameters associated with the object and action.

The scope of this interface is large enough so that each form of control will be described in its own section below after the general interface format is described. Because the interface is so general it is easily extensible by customers.

6.2 Common constants and structures

The `ioctl` objects and actions are presented below. The parameter information supplied with the call varies widely depending on the object. The detailed parameter structure descriptions will be presented in the sections following the interface description when each individual interface is described.

6.2.1 `IOCTL` objects

The following objects are defined. Each will be discussed in a separate section following the general API description.

```
enum ioctlObject
{
  IOCTL_OBJ_FREQ,
  IOCTL_OBJ_CRYPTKEY,
  IOCTL_OBJ_RAW_IO,
  IOCTL_OBJ_RADIO,
  IOCTL_OBJ_AP_JOIN,
  IOCTL_OBJ_ADDR,
  IOCTL_OBJ_CONNOBJ,
  IOCTL_OBJ_FWVER,
  IOCTL_OBJ_PROTOVER,
  IOCTL_OBJ_NVOBJ,
  IOCTL_OBJ_TOKEN
};
typedef enum ioctlObject   ioctlObject_t;
```

6.2.2 `IOCTL` actions

The following actions are defined. They will be discussed as they are relevant in the sections following the general API description.

```
enum ioctlAction
{
  IOCTL_ACT_SET,
  IOCTL_ACT_GET,
  IOCTL_ACT_READ,
  IOCTL_ACT_WRITE,
  IOCTL_ACT_RADIO_SLEEP,
  IOCTL_ACT_RADIO_AWAKE,
  IOCTL_ACT_RADIO_SIGINFO,
  IOCTL_ACT_RADIO_RSSI,
  IOCTL_ACT_RADIO_RXON,
  IOCTL_ACT_RADIO_RXIDLE,
  IOCTL_ACT_RADIO_SETPWR,
  IOCTL_ACT_ON,
  IOCTL_ACT_OFF,
  IOCTL_ACT_SCAN,
  IOCTL_ACT_DELETE
};
typedef enum ioctlAction   ioctlAction_t;
```

[1] The '`ioctl`' terminology is meant to convey the classic notion of application control of non-user space entities at or near the hardware level. The interface does not follow the classic *form* of system `ioctl` calls.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

6.3 SMPL_Ioctl()

6.3.1 Description

This is the single format taken by all `ioctl` calls.

6.3.2 Prototype

```
smplStatus_t SMPL_Ioctl(ioctlObject_t obj, ioctlAction_t act, void *val)
```

6.3.3 Parameter Details

PARAMETER	DESCRIPTION
`obj`	Object of the action requested.
`act`	Action requested for the specified object.
`val`	Pointer to parameter information. May be input or output depending on action. May also be null if object/action combination requires no parametric information.

All instances of '`val`' in calls should be by reference, i.e., a true pointer. Do **not** cast the value of '`val`' to `void *`. The internal code dereferences the argument as if it were a pointer to the object. This can be inconvenient for a simple argument but has the advantage that the interface is completely consistent.

6.3.4 Return

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Operation successful.
`SMPL_BAD_PARAM`	`ioctl` object or `ioctl` action illegal.

Additional return values depend on object specified. These values will be described in the following sections.

6.4 `IOCTL` object/action interface descriptions

6.4.1 Raw I/O

6.4.1.1 Support structure definitions

The following structures support this object:

```
typedef struct
{
  uint8_t  addr[NET_ADDR_SIZE];
} addr_t;

typedef struct
{
  addr_t   *addr;
  uint8_t  *msg;
  uint8_t   len;
  uint8_t   port;
} ioctlRawSend_t;

typedef struct
{
  addr_t  *addr;
  uint8_t *msg;
  uint8_t  len;
  uint8_t  port;
  uint8_t  hopCount;
} ioctlRawReceive_t;
```

6.4.1.2 Interface details

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

This object permits sending to and receiving from arbitrary destination address/port combinations. Normally applications must have established peer connection using the linking scheme. This object permits unconditional communication. This support is used extensively by the **NWK** layer itself.

Note that this interface requires the caller to supply a complete Application address (device address and port number) not a Link ID as would be done from the application.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_RAW_IO	IOCTL_ACT_READ	**ioctlRawReceive_t**	When executed returns the payload for the oldest frame on the specified port. It is similar to a SMPL_Receive() call except that additional information is available from the received frame.
	IOCTL_ACT_WRITE	ioctlRawSend_t	Sends the enclosed payload to the specified address/port combination.

6.4.1.3 Return

6.4.1.3.1 IOCTL_ACT_WRITE

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Transmission successful.
SMPL_NOMEM	No room in output frame queue.
SMPL_TX_CCA_FAIL	CCA failure.

6.4.1.3.2 IOCTL_ACT_READ

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Frame for the Port found. Contents of 'msg' and 'len ' are valid.
SMPL_NO_FRAME	No frame available.

6.4.2 Radio Control

Some simple radio control features are currently available. At this time this interface does not support direct access to the radio configuration registers.

6.4.2.1 Support structure definitions

```
typedef int8_t rssi_t;

typedef struct
{
  rssi_t  rssi;
  uint8_t lqi;
} rxMetrics_t;

typedef struct
{
  linkID_t     lid;   /* input: port for which signal info desired */
  rxMetrics_t  sigInfo;
} ioctlRadioSiginfo_t;
```

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

```
enum ioctlLevel
{
  IOCTL_LEVEL_0,
  IOCTL_LEVEL_1,
  IOCTL_LEVEL_2
};

typedef enum ioctlLevel ioctlLevel_t;
```

6.4.2.2 Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_RADIO	IOCTL_ACT_RADIO_SLEEP	**NULL**	Done before putting the MCU to sleep. Does a disciplined state change to the radio. Saves any radio registers necessary.
	IOCTL_ACT_RADIO_AWAKE	**NULL**	Done after MCU wakes up. Restores any radio registers necessary.
	IOCTL_ACT_RADIO_SIGINFO	**ioctlRadioSiginfo_t**	Get the signal strength information for the last frame received on the specified port.
	IOCTL_ACT_RADIO_RSSI	**rssi_t**	Get current RSSI value
	IOCTL_ACT_RADIO_RXON	**NULL**	Place radio in receive state
	IOCTL_ACT_RADIO_RXIDLE	**NULL**	Place radio in idle state to conserve power
	IOCTL_ACT_RADIO_SETPWR*	**ioctlLevel_t**	Set output power level.

* Enabled with **EXTENDED_API** build time macro definition.

6.4.2.3 Return

6.4.2.3.1 `Null` object

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.2.3.2 `ioctlRadioSiginfo_t` object

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Receive metric information valid.
SMPL_BAD_PARAM	No valid connection information for Link ID specified in parameter structure.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

6.4.2.3.3 `rssi_t` object

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	RSSI value valid. This call always succeeds.

6.4.2.3.4 `ioctlLevel_t` object

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Specified power level valid and set.
SMPL_BAD_PARAM	Invalid power level specified.

6.4.3 Access Point Join Control

To add some control over the ability of a device to gain access to the SimpliciTI network the protocol uses tokens to both join a network and to create peers by linking. Additional control is provided by allowing an Access Point to exclude the processing of Join frames unless the context is set to permit such processing. The idea is that if the device cannot join then it cannot obtain the proper link token for that network so it will not be able to link with any other devices.

6.4.3.1 Interface Details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_AP_JOIN	IOCTL_ACT_ON	**NULL**	Permit processing of Join frames.
	IOCTL_ACT_OFF	NULL	Ignore Join frames.

6.4.3.2 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.4 Device Address Control

This interface permits the application to override the build-time device address setting. If the application generates a device address at run time this interface is used to set that address. The setting of the address **must** occur before the call to **`SMPL_Init()`**. Otherwise the build-time address will prevail. Once the address is set under either condition (pre-initialization `ioctl` call or through **`SMPL_Init()`**) the address cannot be changed.

6.4.4.1 Supporting structure definition

```
typedef struct
{
  uint8_t  addr[NET_ADDR_SIZE];
} addr_t;
```

6.4.4.2 Interface details

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_ADDR	IOCTL_ACT_SET	**addr_t**	Sets address to value pointed to.
	IOCTL_ACT_GET	addr_t	Returns address in address pointed to.

6.4.4.3 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.5 Frequency Control

The current logical channel can be set and retrieved with this interface. A scan can also be requested. All of these interfaces are used by **NWK** in support of Frequency Agility.

6.4.5.1 Supporting structure definitions

```
typedef struct
{
  uint8_t logicalChan;
} freqEntry_t;

typedef struct
{
  uint8_t      numChan;
  freqEntry_t *freq;
} ioctlScanChan_t;
```

6.4.5.2 Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_FREQ	IOCTL_ACT_SET	**freqEntry_t**	Sets logical channel to value pointed to.
	IOCTL_ACT_GET	**freqEntry_t**	Returns logical channel in address pointed to.
	IOCTL_ACT_SCAN	**ioctlScanChan_t**	Scans for replies on all logical channels. Channel numbers on which replies were received are returned in the freqEntry_t array pointed to.

6.4.5.3 Return

6.4.5.3.1 IOCTL_ACT_SET

STATUS	DESCRIPTION
SMPL_SUCCESS	Operation successful.
SMPL_BAD_PARAM	Requested logical channel number is out of range.

6.4.5.3.2 IOCTL_ACT_GET

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.5.3.3 IOCTL_ACT_SCAN

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds. However, the channel count in the returned parameter structure can be 0 which means that no channels were found. Caller should be sure to check the 'numChan' channel count element

6.4.6 Connection Control

Currently the following interface removes the connection entry for the specified Link ID. It does not tear down the connection by alerting the peer that the local connection is destroyed.

6.4.6.1 Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_CONNOBJ	**IOCTL_ACT_DELETE**	linkID_t	Deletes local connection from the connection table that is specified by the link ID pointer.

The Link ID SMPL_LINKID_USER_UUD is not a valid object for this call.

6.4.6.2 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	Operation successful.
SMPL_BAD_PARAM	Link ID is SMPL_LINKID_USER_UUD or no connection information for specified Link ID

6.4.7 Firmware Version

6.4.7.1 Supporting definitions

```
#define SMPL_FWVERSION_SIZE   4
```

6.4.7.2 Interface details

The firmware version that is running can be retrieved. It is a read-only (Get) object.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_FWVER	**IOCTL_ACT_GET**	uint8_t	Retrieves the current firmware version as a byte array.

The firmware version is an array of size SMPL_FWVERSION_SIZE that has the following format:

Byte	Contents
0	Major release number
1	Minor release number
2	Maintenance release number
3	Special release number

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

The values in each byte are binary.

6.4.7.3 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.8 Protocol Version

The protocol version can be used to determine interoperability context or to deny access. It is used during both the Join and Link negotiation. Currently the Join or Link is denied if the versions do not match. Backward compatibility could be implemented under some conditions.

6.4.8.1 Interface details

The current protocol version is read-only.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_PROTOVER	**IOCTL_ACT_GET**	uint8_t	Protocol version.

6.4.8.2 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.9 Non-volatile Memory Object

This object provides direct access to the current connection object. This object contains the context required to establish, maintain, and restore all peer connections. Other information is also kept such as store-and-forward client information if the device is an Access Point. An application can protect against reset conditions by saving and restoring this context appropriately.

The interface provides access to the object by providing an object version value, a length, and a pointer to the object. It is intended that the caller treat the object as a monolithic object and simply save or restore it as a single entity. The version and length information is supplied to help both with local handling and sanity checks when restoring the object.

This interface provides a **GET** action only. Application must do its own sanity checks. When saving a context the length and version elements in the ioctl object should be saved in addition to the monolithic NV object. When restoring a context the application should do a **GET** and then be sure that the object version and length elements match those that were previously saved.

This feature is enabled with **EXTENDED_API** build time macro definition.

6.4.9.1 Supporting structure definitions

```
typedef struct
{
  uint8_t    objVersion;
  uint16_t   objLen;
  uint8_t  **objPtr;
} ioctlNVObj_t;
```

6.4.9.2 Interface details

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_NVOBJ	**IOCTL_ACT_GET**	ioctlNVObj_t	Returns the version and length and a pointer to the connection context.

If the 'objPtr' element is null only the NV object version and length objects are populated.

Note that this interface provides (dangerous) direct access to the connection context area in memory. Care should be taken by applications the not disturb this memory or manipulate the contents directly.

6.4.9.3 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	
SMPL_BAD_PARAM	An action other than IOCTL_ACT_GET was specified.

6.4.10 Network Access Tokens

An interface is provided to get and set the two network access control tokens, the Join token and the Link token.

This feature is enabled with **EXTENDED_API** build time macro definition

6.4.10.1 Supporting definitions

```
enum tokenType
{
  TT_LINK,      /* Token Type is Link */
  TT_JOIN       /* Token Type is Join */
};

typedef enum tokenType tokenType_t;

/* If either token ever changes type a union will make things easier. */
typedef union
{
  uint32_t linkToken;
  uint32_t joinToken;
} token_t;

typedef struct
{
  tokenType_t  tokenType;
  token_t      token;
} ioctlToken_t;
```

6.4.10.2 Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_TOKEN	IOCTL_ACT_GET	ioctlToken_t	Get the value of the specified token into the 'token' object
	IOCTL_ACT_SET	ioctlToken_t	Set the value of the specified token from the 'token' object.

6.4.10.3 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	
SMPL_BAD_PARAM	A token other than TT_LINK or TT_JOIN or an action other than IOCTL_ACT_GET was specified.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

7. Callback Interface

7.1 Introduction

A single callback may be registered during initialization by providing a function pointer as an argument to the initialization call (See Section 3.3). The function must be supplied by the application programmer.

7.2 Callback function details

7.2.1 Description

The callback (if registered) is invoked in the receive ISR thread when the frame received contains a valid application destination address.

7.2.2 Prototype

```
uint8_t sCallBack(linkID_t lid)
```

7.2.3 Parameter details

PARAMETER	DESCRIPTION
`lid`	The Link ID of the connection bound to a received frame.

The parameter in the callback when invoked will be populated with the Link ID of the received frame. This is the way the callback can tell which peer has sent a frame and possibly requires service. The special Link ID `SMPL_LINKID_USER_UUD` is a valid parameter value in this context.

A call to `SMPL_Receive()` using the supplied Link ID is guaranteed to succeed[2]. This is the only means by which the frame can be retrieved.

7.2.4 Return

The callback must either 0 or non-zero. This value is the responsibility of the application programmer.

If the function returns 0 the received frame is left in the input frame queue for later retrieval in the user thread. This is the recommended procedure. The callback can simply set a flag or otherwise store the information about the waiting frame. The actual reference to `SMPL_Receive()` should be done in the user thread.

If it returns non-zero the frame resource is released for reuse immediately. This implies that the callback has extracted all valid information it requires.

[2] The success is guaranteed unless the frame is deleted due to the LRU policy for managing the input frame queue. This can happen if the referenced frame is not retrieved in a timely manner.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

8. Extended API

8.1 Introduction

If the macro **`EXTENED_API`** is defined over the entire project build[3] additional API symbols are enabled. These are described in the Sections that follow. The symbols are not enabled by default to save code space. If the macro is defined all the symbols are included.

8.2 SMPL_Unlink()

8.2.1 Description

This API is used to tear down a connection in a disciplined manner. Disabling the connection consist of two actions. First, the local connection is unconditionally disabled. After this call any further references to the relevant Link ID will result in a return of **`SMPL_BAD_PARAM`**.

Second, a message is sent to the peer to inform the peer that the connection is being terminated. The calling thread will wait for a reply. If a reply is received it contains the result of the connection termination attempt on the peer. If a reply is not received the return from the call so indicates.

There is no guarantee that the message sent to the peer will be received. If the peer does not get the connection termination frame it must have some independent means to determine that the connection has been terminated

8.2.2 Prototype

```
smplStatus_t SMPL_Unlink(linkID_t lid)
```

8.2.3 Parameters

PARAMETER	DESCRIPTION
`lid`	The Link ID of the connection to be disabled.

8.2.4 Return

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Unlink successful on both peers.
`SMPL_BAD_PARAM`	Link ID not found.
`SMPL_TIMEOUT`	No response from peer.
`SMPL_NO_PEER_UNLINK`	Peer did not have a Connection Table entry for specified connection.

8.3 SMPL_Ping()

8.3.1 Description

[3] This is done within the IAR IDE by defining the macro in the **`smpl_nwk_config.dat`** project file.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

This API implements the NWK Ping application on behalf of the User application. It pings the device associated with the peer specified. Note that it does *not* ping the peer itself but rather the device on which the peer is hosted. It is roughly equivalent to the ICMP application in the TCP/IP suite.

It is provided as a convenience for the User applications. It can be used to see if the hosting device is there. Since it does not talk to the peer itself it does not verify that the peer is there, but only that the device hosting the peer is there.

This API has the convenient side effect. If Frequency Agility is enabled it will scan the channels in the channel table if a reply is not received on the current channel. So, an application can discover a changed channel for free instead of implementing its own scan channel logic.

8.3.2 Prototype

```
smplStatus SMPL_Ping(linkID_t lid)
```

8.3.3 Parameters

PARAMETER	DESCRIPTION
`lid`	The Link ID of the peer whose device should be pinged.

8.3.4 Return

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Ping succeeded.
`SMPL_TIMEOUT`	No response from peer.

8.4 SMPL_Commission()

8.4.1 Description

This API is used to statically create a connection table entry. It requires detailed knowledge of the objects in the connection table. If both peers are (correctly) populated using this API a connection can be established without an explicit over-air linking transaction.

When used to create static connections User must know in advance the SimpliciTI address of the device for each peer. In addition, both local and remote port assignments must be made. The local port number on one device must correspond to the remote port assignment on the other device. They may have the same value but should be unique for each peer on a specific device.

The User port address space is partitioned into static and dynamic portions. The size of the static portion, the portion from which ports using this API must be drawn, is defined by the macro **`PORT_USER_STATIC_NUM`** found in the file **`.\Components\nwk.h`**. The default value is 1. The static port address space starts at `0x3E` and builds down.

Range checks are made on the port assignments but other sanity checks, such as duplicate assignments, are not made.

8.4.2 Prototype

```
smplStatus_t SMPL_Commission(addr_t *peerAddr, uint8_t locPort, uint8_t rmtPort, linkID_t *lid)
```

8.4.3 Parameters

PARAMETER	DESCRIPTION
`peerAddr`	Pointer to address of peer.
`locPort`	Local static port assignment
`rmtPort`	Remote static port assignment.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

lid	Pointer to Link ID object. Value assigned by NWK.

8.4.4 Return

STATUS	DESCRIPTION
SMPL_SUCCESS	Connection successfully created.
SMPL_BAD_PARAM	Bad Link ID pointer (value null) or ports out of range.
SMPL_NOMEM	No room in connection table.

Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

9. Extended support

9.1 Introduction

In addition to the SimpliciTI API there are various support macros and functions available for use by applications. These are for convenience. As application examples were developed support in the form of certain "helper" utilities seemed sensible.

These are described in the following sections. The macros are defined in the file `nwk_types.h`.

9.2 NWK_DELAY()

9.2.1 Description

This macro will implement a synchronous delay specified in milliseconds. It is not accurate so should not be used for time-sensitive applications.

It is used in the application examples for crude switch de-bouncing and delays between LED toggles to indicate application state.

9.2.2 Prototype (macro)

```
NWK_DELAY(uint16_t msDelay)
```

9.2.3 Parameter description

PARAMETER	DESCRIPTION
`msDelay`	Number of milliseconds to delay.

9.2.4 Return

N/A.

9.3 NWK_REPLY_DELAY()

9.3.1 Description

An application can invoke this macro after sending a message to a peer from which it expects an immediate reply. The delay will terminate as soon as the next application frame is received (presumably the expected reply) or when a maximum time has expired. The maximum delay time is computed during initialization of the stack and is scaled by the data rate and the maximum application payload size. It requires no user intervention.

A sample message exchange session using this macro is shown below.

9.3.2 Prototype (macro)

```
NWK_REPLY_DELAY()
```

9.3.3 Parameter description

N/A

9.3.4 Return

N/A

9.3.5 Example of macro usage

This code is incomplete in the sense that variable declarations and result code checks are not shown. However, the symbol references are all conformal.

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

```
/* Time to send a message to peer whose Link ID is 'linkID' */
{
  /* wake up radio */
  SMPL_Ioctl(IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_AWAKE, 0);

  /* Send message */
  SMPL_Send(linkID, &sendMsg, sizeof(sendMsg));

  /* Radio must be in Rx state to get reply. Then back to
   * IDLE to conserve power.
   */
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RXON, 0);
  NWK_REPLY_DELAY();
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RXIDLE, 0);

  /* Get received reply */
  SMPL_Receive(linkID, &rcvMsg, &rcvMsgLen);

  /* radio off */
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SLEEP, 0);
}
```

 Copyright © 2008-2009 Texas Instruments, Inc. All rights reserved.

B.3. Notas para el Desarrollador

SimpliciTI: Simple Modular RF Network Developers Notes

Author: Larry Friedman

Texas Instruments, Inc.
San Diego, California USA

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Version	Description	Date
0.99	Pre-release draft	05/25/2007
1.00	Initial Release	07/02/2007
1.10	Revised to reflect SimpliciTI release 1.0	08/31/2007
1.20	Revised to reflect SimpliciTI release 1.0.4	02/01/2008
1.30	Revised to reflect SimpliciTI 1.0.6.	08/01/2008
1.40	Revised to reflect SimpliciTI 1.1.0.	03/24/2009

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

TABLE OF CONTENTS

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

LIST OF FIGURES

LIST OF TABLES

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

1. Introduction

This document provides information necessary to effectively use the SimpliciTI protocol support. Frequent references are made to source code files since source code is provided for this support.

There are some differences in the implementation depending on the specific radio being used.. The references to the firmware supporting the protocol itself, however, are not hardware dependent.

2. References

[1] *SimpliciTI Specification*, Texas Instruments, 2007

[2] *Application Note on SimpliciTI Frequency Agility*, 2008

[3] *SimpliciTI Application Programming Interface*

[4] *Application Note on SimpliciTI Security*

3. Overview

SimpliciTI is a connection-based peer-to-peer protocol. It supports 2 basic topologies: strictly peer-to-peer and a star topology in which the star hub is a peer to every other device. The Access Point is used primarily for network management duties. It supports such features and functions as store-and-forward support for sleeping End Devices, management of network devices in terms of membership permissions, linking permissions, etc. The Access Point can also support End Device functionality, i.e., it can itself instantiate sensors or actuators in the network. In the star topology the Access Point acts as the hub of the network.

The protocol support is realized in a small number of API calls. These APIs support Customer application peer-to-peer messaging. The association between two applications, called linking, is done at run time. The linking process creates a connection based object through which the application peers can send messages. When a connection is established it is a bi-directional connection. There are provisions for a basic commissioning mechanism as well in which the connection context is populated directly with application layer calls.

4. Hardware configuration

4.1 MCU Interface

The CC1100/CC2500 radios specify a radio-MCU interface. In addition to the SPI communications the interface provides for up to two additional lines that when connected to MCU GPIO pins can be configured to generate interrupts. Though many of the reference designs provide for both of these to be connected, the implementation herein assumes only one of these (GDO2) is connected.

The CC2520 radio-MCU interface also specifies a GPIO line usable for interrupts and an SPI interface for command strobes and data transfer.

The SoC solutions depend on memory-mapped registers for data transfer and command strobes and they supply an interrupt vector location for presenting interrupts.

4.2 Radio configuration

The source file from which the initialization values for the radio are taken was generated by the SmartRF Studio tool from TI. The exported register configuration is included in the code distribution.

In addition in some cases additional register settings are added. These consist of setting registers that are not exported explicitly by the SmartRF Studio tool.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

5. Architecture overview

5.1 Protocol layers

The protocol is in service to an application layer in which the main focus is peer-to-peer communication. The peers would typically be sensor-controller and actuator-controller objects. Direct sensor-actuator peers are supported as well. The protocol makes no distinctions here.

The goal of the protocol from the implementation perspective is to make it simple to link together various arbitrary peer applications.

A schematic of the layering is shown in the following Figure:

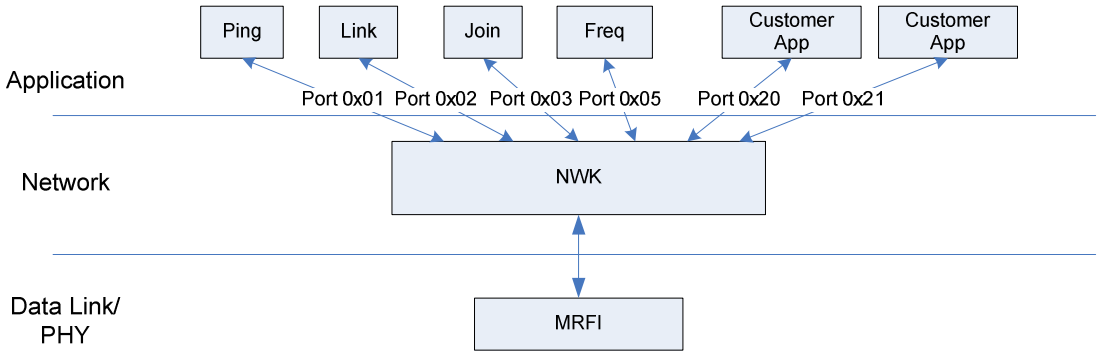


Figure 1: SimpliciTI logical layers

There is no formal **PHY** or Data Link (**MAC/LLC**) Layer. The data are received directly from the radio already framed so the radio performs these functions. The MRFI (Minimal RF Interface) layer encompasses whatever support is necessary to interact with the radio.

There is also an entity (not shown) called Board Support Package (BSP) to abstract the SPI interface from the **NWK** layer calls that interact with the radio. It is not intended to support a general hardware abstraction in service to the applications. Only those services (such as the SPI interface) that are in direct support of the **NWK**-radio interface are supported. As a convenience it also supports LEDs and button/switch peripherals attached to GPIO pins. But no other services are provided such as UART drivers, LCD drivers, or timer services.

The **NWK** layer manages the Rx and Tx queues and dispatches frames to their destination. The destination is always an application designated by a Port number. The **NWK** layer does no frame processing on behalf of applications.

The Ports are similar in spirit to the notion of a TCP/IP port. It is conceptually an extension of the address. The network frame overhead is stripped off and the remaining payload is dispatched to the application that lives on the designated Port.

The **NWK** layer applications are on "well known" ports. These all have values ≤ 0x1F. They are used by the **NWK** layer itself to manage the network. These ports are not intended to be accessed directly by Customer applications.[1] The **NWK** layer applications are not connection-based.

The Customer application Ports are assigned during the Link process by the **NWK**. As a result of a successful link transaction the application receives a handle called the Link IDs. The mapping from Link ID to address is done by

[1] The Ping application is an exception to this. This application is intended primarily for debugging, as it is awkward to use otherwise. The destination address must be known to use this Port (just like the IP Ping) and typically this isn't known at the application layer.

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

the **NWK**. This is similar to the Sockets approach. The application has no responsibility with respect to assigning and maintaining the Port objects.

5.2 NWK applications

The **NWK** applications support network management. Except possibly for Ping these applications are not intended to be part of the Customer's development environment. They are described here to provide a more comprehensive understanding of how SimpliciTI protocol supports the communication s environment.

Application	Port	Description
Ping	0x01	Just like the TCP/IP application. Echoes the received payload back to the sender. Direct addressing only.
Link	0x02	Used to associate two peers on different devices.
Join	0x03	Used when an Access Point is present to gain access to peers.
Security	0x04	Used to exchange security information.
Freq	0x05	Used to manage frequency migration to support frequency agility.
Mgmt	0x06	General use **NWK** management application. Used for example as the poll port.

Table 1: NWK applications

5.3 Peer layer characteristics

There are essentially two SW peer layers in this architecture: **NWK** and Application. As seen in Figure 1 the Application layer is partitioned into two parts: **NWK** applications and Customer applications. Characteristics of each are described in Table *2*:

Layer	Connection Type	Acknowledgment
NWK	Connectionless	No
NKW Applications	Connectionless	Yes[2]
Customer applications	Connected. The connections are bi-directional.	Optional depending on Customer's implementation

Table 2: Peer layer characteristics

For development purposes it is important to note that SimpliciTI in its basic form does not support acknowledgements. The main consequence of this is the need for applications themselves to provide support for the following:

- Segmentation and reassembly for messages larger than the maximum application payload
- Missing data (no **NWK** guaranteed delivery in the form of Transport layer)
- Redundant data (no **NWK** recognition of duplicate frames)

[2] There are some exceptions to this. In general **NWK** application frames are acknowledged but some are not. In any case these exchanges are not visible to the Customer applications.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

5.4 Message acknowledgements

With SimpliciTI Release 1.1.0 some support for acknowledgement is provided. When an application sends a message requesting an acknowledgment, a success (i.e., acknowledgement received) means only that the peer's **NWK** layer received the frame. It does not imply that the peer application itself received the frame.

This is not peer-to-peer acknowledgement and should not be use as sole determination of guaranteed delivery.

See [3] for details on the use of this feature.

6. Protocol overview

The functionality provided by the protocol is simply to provide support for connection-based peer-to-peer communications. The intent is to wrap the fundamental radio portion and remove that domain from the Customer's concern during development.

The functionality is realized in a simple set of API calls available to the Customer's application. The simplicity comes with the price of flexibility. The vision is that the use of this simple, small footprint protocol will be in scenarios that require only limited flexibility.

The following discussion summarizes the mechanisms that are implemented in support of the protocol.

6.1 Topology

The protocol addresses only peer-to-peer topologies. With this protocol there is no formal routing mechanism supported. The Access Point (if present) supplies the support needed to manage the network. This support includes network access and network management functions such as frequency agility.

A star topology is realized by setting the hub to be a peer of each other device on the network.

6.2 SimpliciTI objects

SimpliciTI objects are SW objects. Three SimpliciTI objects are supported: End Devices, Access Points, and Range Extenders. Each is a logical construct so that multiple objects can be realized on a single hardware platform. For example, a platform that contains an Access Point can also support an End Device. However, most End Devices will likely occupy a hardware device either alone or with other End Devices.

6.2.1 End Device

These are the simplest devices. They are the locus of most of the sensors/actuators in the network. The End Devices host the application peers. A hardware platform hosting only End Devices may be battery powered.

6.2.2 Access Point

The Access Point, when present, can act as the star hub in the network. These are always-on devices. Only one Access Point per network is permitted. This can be enforced using constructs discussed below.

Access Points may coexist with End Devices on the same hardware platform. They can host peer applications that realize sensors or actuators, or both on the network.

Access Points run in promiscuous mode will receive all packets within range. In addition to infrastructure support Access Points will replay frames not destined for itself to help extend the range of End Devices.

6.2.3 Range Extender

The Range Extenders are intended to extend the radio range on a network. These are always-on devices. The main function is to repeat frames effectively extending the sphere of influence of the frame sender. Currently networks are limited to 4 Range Extenders.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Although not seen as a common use, Range Extenders may coexist with End Devices on the same hardware platform. They can host peer applications that realize sensors or actuators, or both on the network.

Range Extenders run in promiscuous mode will receive all packets within range.

6.3 Address namespace

A network addresses consist of two parts: a platform hardware address and an application address (Port).

As distributed the hardware address for each device is assigned at build time. A capability is provided to assign the address at run time as well. It must be unique for each device in the network. The management of the hardware address space is up to the Customer. There is no address resolution protocol.

Currently the hardware address space is that spanned by a 4-byte quantity treated as an array of unsigned characters.

The application address (Port) is either well-known and fixed or assigned at run time during the device Link procedure. It is not under Customer SW control.

6.4 Network discipline

In this Section will be a brief overview discussion of how peer-to-peer connections are made and how network member ship is controlled.

More information on the details of the frames involved in the connection processes can be found in [1].

6.4.1 Linking

Linking, supported by the SimpliciTI API, is the means by which an application peer-to-peer connection is established. Links are established in pairs. One application of the pair listens for a link message from the application with which it should be paired. Which application listens and which sends the link message is arbitrary, as the resulting connection is bi-directional.[3] Typically a link session would be instigated by an end user action such as a button press or other physical intervention.

The link message contains a link token (currently a 4 byte object) which is used by the listener to validate the peer. There is a default link token set by the Customer at build time. Additional constraints can be added if the network has an Access Point. See Section 6.4.2.

Links are logical entities. A single platform may support multiple peer-to-peer links. These may be multiple links between the same two applications, multiple application pairs with single links, or any combination. The pairs may be on any two distinct platforms. SimpliciTI does not support links between two applications on the same platform.

The number of links on a device is limited only by RAM and the port address space.

6.4.2 Joining

The Join action is supported only when the network contains an Access Point. Joining is *not* supported by a specific SimpliciTI API but is a side effect of the initialization call. It is the process by which a platform gains access to the network from an Access Point. Joining is the first action on the part of a device after initialization and before any other actions are taken.

The platform that wants to join a network sends a Join message as part of the SimpliciTI initialization. The message contains a Join token (currently a 4 byte object) which is set by the Customer at build time. The Join token can be used to ensure that two Access Points do not both respond to a new device trying to join a network. If the Join token matches the token expected by the Access Point the Access Point will reply with network information that the

[3] If one of the applications is on a Tx-only platform the assignment of roles becomes constrained.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

platform will require in order to interact properly on the network. Currently this information includes the Link token for the network (see Section 6.4.1).

The use of the unique Join token might be awkward if after-sale devices are added to the network. It would require the new device to somehow know the network's Join token at the point of installation. To deal with this scenario there is an additional capability to set the Join context in the Access Point to be either active or inactive. The context could be activated on the Access Point in the same way a link context is set (button press, etc.) and a default Join token could be used. By default the Join context is always active.

The act of joining a network does not provide anything other than infrastructure information such as the Link token to the joining device. The Access Point does not track joining devices except in the case of a polling device for which the Access Point must supply store-and-forward support (See Section 6.4.3).

6.4.3 Sleeping End Devices

Sleeping End Devices can take one of two approaches to getting messages. One type polls the Access Point to see if any messages are waiting. The second type listens for activity and if there is activity it stays awake and looks for frames destined for it.

The type of sleeping device is configured at build time.

6.4.3.1 Polling devices

If a sleeping End Device is configured as a polling device it is recognized as such by the Access Point when the device joins the network. At this time the Access Point reserves resources to support the device. All messages addressed to the sleeping device on a non-network Port are held by the Access Point. Broadcast messages are not held.

When the sleeping device awakens and does a receive call the call results in a polling message to the Access Point. This polling message specifies the Port being queried and is sent on the Management Port. The Access Point sends the oldest frame on the queried Port destined for the polling device. If none are being held the Access Point sends a frame with no payload to the queried port.

In the current implementation each Port must be polled separately and each Port must be polled until there are no more messages on that Port.

7. Application programming

This section presents information on the SimpliciTI implementation that will aid in the proper construction of a Customer application using the SimpliciTI capabilities.

7.1 Development environment

There are two development environments for SimpliciTI. First is IAR Embedded Workbench. this environment can be used for both the 8051 core SoC targets and the dual chip (MSP430 + radio) solutions. The Texas instruments Code Composer Essentials v. 3.1 (CCE) IDE also supports SimpliciTI for the dual chip (MSP430 + radio) targets.

Except for IDE-dependent files (such as the linker script which is the default for the target) there are no known IDE-dependent constructs or keywords used in the source code.

7.2 Hardware abstraction

Only a bare minimum hardware abstraction is provided with the distribution. This abstraction (BSP) supports the radio interface. It also includes an abstraction to support up to 8 LEDs and 8 switches on the target board.

There is no support for UARTs, timers, LCDs or other potential peripherals or on-chip resources. These implementations are left completely to the Customer.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

7.3 MCU Resources

Other than flash for code space and constants, and RAM, the SimpliciTI implementation uses no other resources. It does not currently depend on dynamic memory allocation so it uses no heap memory. All memory allocations are either static or are represented as automatics and therefore use the stack. The **`const`** memory type is used when possible to save RAM. The implementation *does* optionally use one timer resource though the least intrusive timer resource on the MCU is chosen when the hardware timer resource is sued.

The run-time context is not stored in any persistent memory on either the MCU or the radio. With some minor exceptions that are handled in the code, the initial radio targets do not lose run-time context when in sleep mode. Most of the SRAM is maintained and those critical values that are not maintained are restored.

7.4 Threading model

In general the SimpliciTI API is designed to run in conjunction with the threading model of the Customer application. The operation of the protocol is not constructed as an independent task requiring its own context. Since the code runs in the existing threading context the implementation requires no access to a scheduler or any other OS construct.

The SimpliciTI API should **not** be considered thread-safe or re-entrant.

SimpliciTI provides a general callback capability for received frames. This callback runs in the receive ISR thread so efficient use of the callback is encouraged.

7.4.1 Peer-to-peer I/O

After initialization the peer applications are linked using a pair of sequenced calls in the API. The result of these calls on each end is a Link ID. This is a handle, like a socket except that the binding is done automatically. The Link ID is used to reference the connection subsequently. After the linking step the may read from and write to the peer by referencing the Link ID Basically the API consists of a read (Receive) call and a write (Send) call.

7.4.1.1 Input/receive

On devices that do not sleep the receipt of a frame by the radio triggers an interrupt to the MCU. The frame is read out of the radio Rx FIFO and stored in user space in the MCU in a queue of received frames. If the input frame queue is full when a frame is received the oldest frame in the queue is dropped. The interrupt thread is then released. This is a relatively efficient process.

When the application level read is done on the specified Link ID the input frame queue is examined for any frames waiting for that Link ID. If a frame is waiting the application payload is returned to the caller. Otherwise the caller receives an indication that there are no data. In essence a read is a poll of the input frame queue for any frames waiting for the designated Link ID. Payloads are returned in FIFO order.

If the device is a sleeping/polling device the application layer read kicks off a poll query to the Access Point. This is invisible to the caller. The thread is then held and waits for the reply from the Access Point. If the Access point forwards a frame it is then passed back to the caller. If the Access point sends a frame with no payload it is interpreted as a simple acknowledgement and the return to the caller looks like a poll that returns with no payload. Because this scenario holds the thread until the reply from the Access Point it is less efficient than the non-sleeping case that simply examines the input queue for data.

7.4.1.2 Output/send

The sending scenario is implemented as a synchronous call that does not return until the frame is transmitted by the radio. This design prevents unintentional termination a transmit sequence by removing radio power before the frame is sent.

If the Tx thread cannot get access to the radio channel to transmit the frame the caller receives a return code accordingly and can retry later. There is some robustness in the network layer, as there is some degree of recovery

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

attempted if access to the channel cannot be obtained immediately. But this is minimal. It is left to the Customer application to decide how to deal with this scenario since the SimpliciTI **NWK** does not support guaranteed delivery (see Table 2). Only the application knows how important it is to send the frame. The power consumption vs. communications reliability tradeoff is left to the Customer application.

7.4.2 NWK application threads

Since the **NWK** application threads (those that service the **NWK** application ports) are not part of the Customer application context and use no scheduling services they must be handled differently.

The Join, Link, and Mgmt polling frames each require a response from the target device. At the application layer this makes them acknowledged frames. To ensure that the application has the opportunity to complete the task (i.e., joining or linking) the application will wait for the acknowledgment to arrive. Since callbacks are not used and there is no scheduler involved the sending application holds the calling thread until the reply is received. So, for joining, linking, and polling the sending device may take a "long time" (milliseconds) before completing.

The receiving device for these same applications also handles everything in one thread. In this case, the receive and acknowledge (Tx) are done in the same thread. In this case, the thread is an ISR thread so it is possible for interrupts to be disabled for a "long time" (milliseconds) when handling such a frame.

Note that while these threads can be "long" they do not occur very often. Joining occurs only at startup. Linking typically occurs at startup, though can occur anytime. Polling may occur more often but is still relatively infrequent.

7.5 Object model

SimpliciTI has no formal object model. There are no profiles or other abstractions that define peer application characteristics. The focus of the protocol is to set up peer-to-peer connections and support messaging over air to and from each peer. The object model is created by the Customer, as it is implicit in the application layer protocols set up between peers.

7.6 Sample SimpliciTI transactions

The following is a sample SimpliciTI session. It is intended to convey a general sense of how SimpliciTI peers are linked. The presence of an Access Point is optional. If it is not present the default Link tokens must match in the two End Devices or the listener will reject the Link message by simply not responding.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

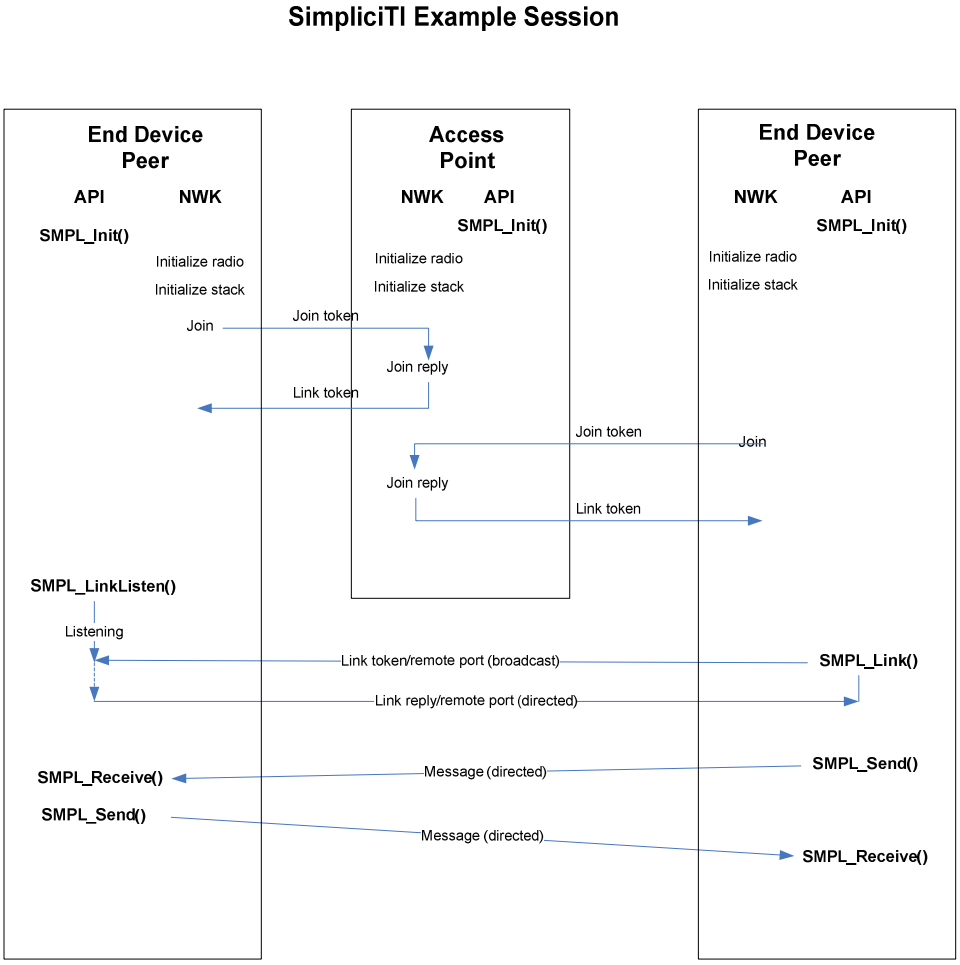


Figure 2: Sample SimpliciTI peer-to-peer session

8. Network access control

There are various means of controlling access to specific network. There is likely a need to guard against rogue devices whether their presence is purposeful or accidental

There are a number of mechanisms built into SimpliciTI to allow various forms of access control. Two of these take advantage of the added control that is possible when an Access Point is part of the configuration. Each of these is discussed below.

8.1 Join token

If an Access Point is part of the configuration then it supports the Join **`NWK`** application. The main purpose of this application is to provide hints to the remaining devices joining the network as to parameters of that network. This would include Link token and encryption key, for example. The idea is that without properly joining the network (i.e., knowing the Link token and key) a rogue device could not interact on the network. Devices need to know the Join token for the specific network to find out the other network parameters.

Customers can configure networks to have different Join tokens, or perhaps, different tokens for different products or types of applications. Join response could be modified as well. For example, the Access Point could assign different Link tokens to different network devices. The Join **`NWK`** application payload could be modified to provide more or different information than the default.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

8.2 Access Point Join context

It might be necessary to actively invoke the Join context on an Access Point. For example if two unrelated Access Points are in proximity to a joining device (e.g., two neighboring houses) additional control might be required to constrain which network it joins.

By default Access Points will always accept a Join frame with the correct Join token. As an added control on access a Customer application can set the Join context of an Access Point. That is, it can be set to reject Join frames unless they are set to actively permit such frames. This is done using the **SMPL_Ioctl()** API.

8.3 Link token

If an Access Point is part of the configuration then it supports the Join **`NWK`** application. The main purpose of this application is to provide hints to the remaining devices joining the network as to parameters of that network. One of these network parameters is the Link token. The idea is that for each network the Link token would be unique, or at least controlled. Two devices subsequently linking must use the correct token or they cannot link.

If there is no Access Point a link token is still required for establishing a connection between two peers. In this case the link transaction is governed by the default link token, the value supplied at build time.

Using the Link token scheme provides an alternative to explicit physical binding in which the physical co-location of two devices provides the security needed to exclude unwanted devices.

8.4 Encryption

Encryption can be used to augment network access control. Encryption is implemented as of SimpliciTI Release 1.1.0. See Reference [4] on SimpliciTI Security implementation.

9. System configuration

Reference [1] summarizes the build-time configurable values needed by SimpliciTI. These are repeated here with additional discussion

There are two configuration file types used when building SimpliciTI devices. One file applies to the network in general and supplies macro values for all devices built for that particular network The other file is device –specific. Each file type is discussed below.

9.1 General network configuration

The file **`smpl_nwk_config.dat`** contains the following network-wide macro definitions:

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Macro	Default value	Comments
MAX_HOPS	3	Frames replayed only 3 times maximum
MAX_HOPS_FROM_AP	1	Maximum distance from an AP. Used by poll and poll response to limit network replay traffic.
MAX_APP_PAYLOAD	10	Absolute maximum is based on size of Rx and Tx FIFOs (50 for CC2500/CC1100 class including SoCs and 111 for IEEE radios CC2430/CC2520).
DEFAULT_JOIN_TOKEN	0x01020304	Change this to provide some access security
DEFAULT_LINK_TOKEN	0x05060708	Should be changed by APs on AP networks. Customer should modify **nwk_join.c:generateLinkToken()**
FREQUENCY_AGILITY	Not defined	When defined enabled support for Frequency Agility. Otherwise only the first entry in the channel table is used.
APP_AUTO_ACK	Not defined	Adds support for application level auto acknowledgment. Requires Extended API
EXTENDED_API	Not defined	Enables **SMPL_Unlink()**, **SMPL_Ping()** and **SMPL_Commission()**.
NVOBJECT_SUPPORT	Not defined	Adds support for gerting and setting connection context for saving across resets.
SMPL_SECURITY	Not defined	Enables security infrastructure

Table 3: General network configuration macros

9.2 Device specific configuration

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

Macro	Default value	Comments
NUM_CONNECTIONS	4	Number of connections supported. Each connection supports bi-directional communication. Access Points and Range Extenders can set this to 0 if they do not host End Devices.
SIZE_INFRAME_Q	2	Two is probably enough for an End Device. Little bit larger for REs and APs. AP needs larger input frame queue if it is supporting store-and-forward clients because the forwarded messages are held here.
SIZE_OUTFRAME_Q	2	The output frame queue can be small since Tx is done synchronously. If an Access Point device is also hosting an End Device that sends to a sleeping peer the output queue should be larger because the waiting frames in this case are held here. Actually 1 is probably enough otherwise.
THIS_DEVICE_ADDRESS	{0x78,0x56,0x34,0x12}	This device's address. The first byte is used as a filter on the CC1100/CC2500 radios so THE FIRST BYTE MUST NOT BE either 0x00 or 0xFF. Also, for these radios on End Devices the first byte should be the least significant byte so the filtering is maximally effective. Otherwise the frame has to be processed by the MCU before it is recognized as not intended for the device. APs and REs run in promiscuous mode so the filtering is not done. This macro initializes a static const array of unsigned characters of length **NET_ADDR_SIZE** (found in nwk_types.h).
Range Extender		
RANGE_EXTENDER		Device type declaration for Range Extenders
End Device		
END_DEVICE		Device type declaration for End Devices
RX_POLLS	Not defined	Define RX_POLLS of the device is a polling End Device.
Access Point		
ACCESS_POINT		Device type declaration for Access Points
NUM_STORE_AND_FWD_CLIENTS	3	Number of store-and-forward clients supported.
AP_IS_DATA_HUB	Not defined	If this macro is defined the AP will be notified through the callback each time a device joins. The AP should be running an application that listens for a link message on receipt of this notification. The ED joining must link immediately after it receives the Join reply.

Table 4: Device-specific configuration macros

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

10. General information and caveats

The following are in no particular order.

1. **CC25xx/CC11xx radios only:** SimpliciTI uses exported register settings from the SmartRF Studio configuration tool. This tool exports register values for both over-the-air RF settings, as well values for control register settings. SimpliciTI uses the values for over-the-air RF settings but must ignore most of the control register settings. The control settings are affect functionality so they must not be overridden. The following table lists registers that are directly controlled by software:

Register	Comments
`IOCFG0`	GPIO configuration. Controlled exclusively by software.
`IOCFG2`	GPIO configuration. Controlled exclusively by software.
`MCSM0`	State machine configuration. The value of PO_TIMEOUT is extracted from SmartRF Studio output, all other fields are controlled by software.
`MCSM1`	State machine configuration. Controlled exclusively by software.
`PKTLEN`	Packet length. Controlled exclusively by software.
`PKTCTRL0`	Packet control register. The value of WHITE_DATA is extracted from SmartRF Studio output, all other fields are controlled by software.
`PATABLE0`	SmartRF Studio does not currently export this value. If a future revision does export this value, it will be used instead of the built-in software default.
`CHANNR`	Channel number. The value exported by SmartRF Studio is used. However, this can be overridden by defining MRFI_CHAN at the project level as equal to the desired channel number.

Table 5: CC1100/CC2500 register setting exceptions

2. See [2] for details on the Frequency Agility feature. See Section 3.1 in that document for details about how to modify the channels in use for this feature.
3. The **SMPL_LinkListen()** call blocks for a relatively long period of time on the assumption that it is executed in temporal contiguity with the **SMPL_Link()** call on the peer. The blocking time may be modified by changing the appropriate commented macro definitions in the file **`nwk_api.c`**.
4. Arrival order for received frames is maintained. If the frame queue is filled and a new frame arrives the oldest frame is discarded to make room.
5. When an application does a **SMPL_Receive()** it will receive the payload from the oldest frame available for the specified link ID.

Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

6. When frames are forwarded to a store-and-forward client they will be forwarded in the order received.
7. Broadcast frames and **NWK** application frames are not saved on behalf of store-and-forward clients.
8. If the Access Point itself hosts an application that sends to a polling device, these frames will be sent after all frames forwarded from other devices instead of in the order in which they were made available.
9. Access Points do not remember the addresses of joined devices. This means that Access Points will replay frames it sees regardless of source. (The same is true for Range Extenders.) It is up to the peer application to ignore rogue messages. The use of the Join and Link tokens can mitigate interference from non-member devices by helping to prevent connections to unauthorized devices but it is no guarantee.
10. Frames to user applications are partially validated at the **NWK** layer by matching the destination port and source address to connection table entries. The received frame must pass this assessment or it is discarded. Otherwise there is no way to remove them from the input frame queue since there will be no application trying to retrieve these frames.

 Copyright © 2007-2009 Texas Instruments, Inc. All rights reserved.

B.4. Tabla de Canales de Frecuencia

SimpliciTI Channel Table Information

	Wi-Fi		IEEE Radios	CC2500/10/11		CC1100/CC1110		CC1100E (470MHz)		CC1100E(950MHz)	
Freq(GHz)	Channels	Freq(GHz)	Channel Numbers	Channel Numbers	Peaks (GHz)	Freq(MHz)	Channel Numbers	Freq(MHz)	Channel Numbers	Freq(MHz)	Channel Numbers
		2.405	11								
2.412	1	2.410	12			902		480		955	
2.417	2	2.415	13			904		482		956	5
2.422	3	2.420	14			906	20	484	20	957	10
2.427	4	2.425	15	3	2.4257	908		486		958	15
2.432	5	2.430	16			910		488	40	959	20
2.437	6	2.435	17		2.4353	912	50	490		960	
2.442	7	2.440	18			914		492	60		
2.447	8	2.445	19			916		494			
2.452	9	2.450	20	103	2.4508	918	80	496	80		
2.457	10	2.455	21			920		498			
2.462	11	2.460	22			922		500			
		2.465	23			924	110				
		2.470	24			926					
		2.475	25	202	2.4755	928					
		2.480	26	212	2.4807						

Wi-Fi Inteference bands
Frequency channel used in SimpliciTI

Notes: For CC2500, CC2510 & CC2511

In SmartRF Studio
Set RF (base) frequency to 2425.0 MHz
Set Channel (spacing) to 250 KHz

Notes: For CC1100 & CC1110

In SmartRF studio
Set RF (base) frequency to 902 MHz

Notes: For CC1100E

In SmartRF Studio
Set RF (base) frequency to 480 MHz (470MHz Band)
and 955 MHz (950MHz Band)
Set Channel (spacing) to 200KHz

Apéndice C

Texas Instruments® ez430-RF2500

C.1. Esquemáticos

C.2. PCB Design

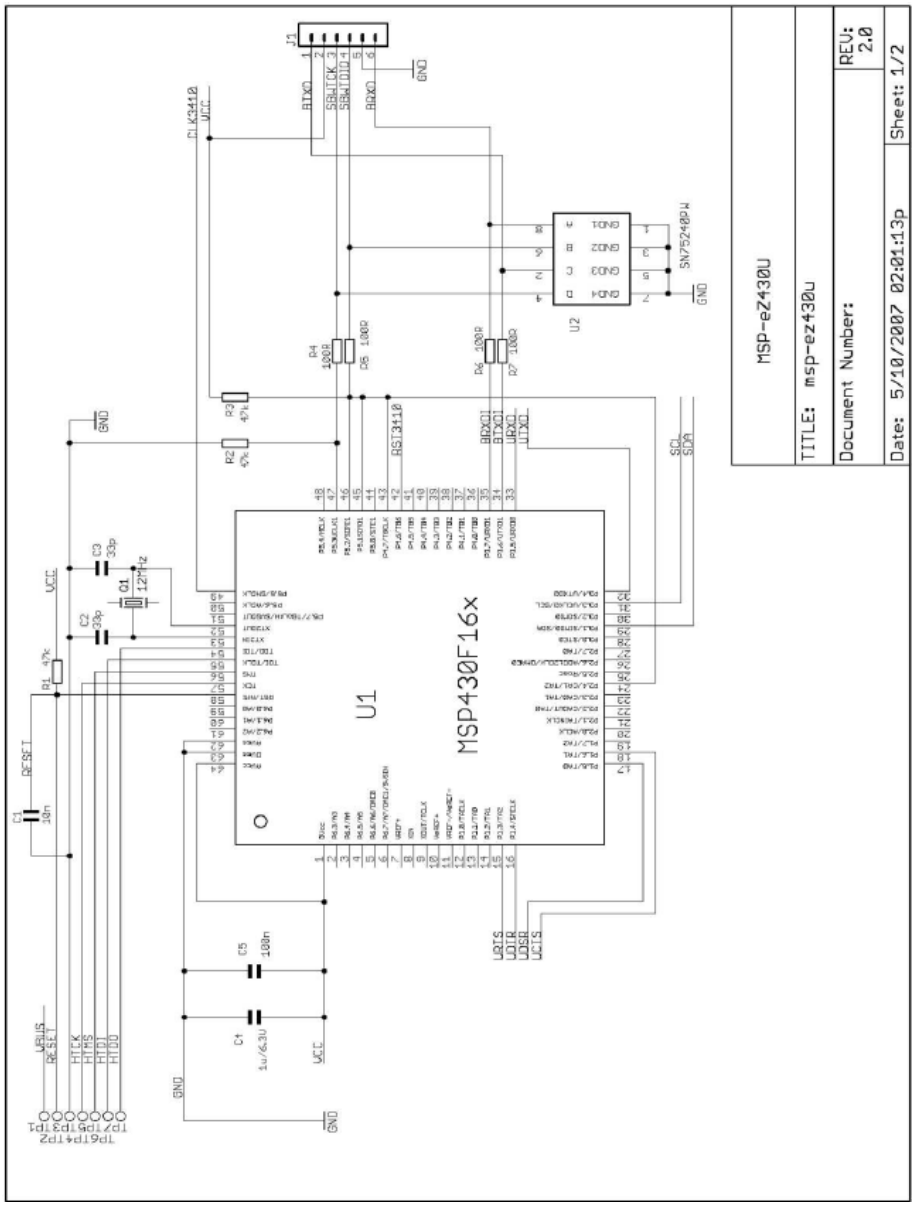


Figura C.1: Diagrama esquemático del ez430-RF2500 USB.

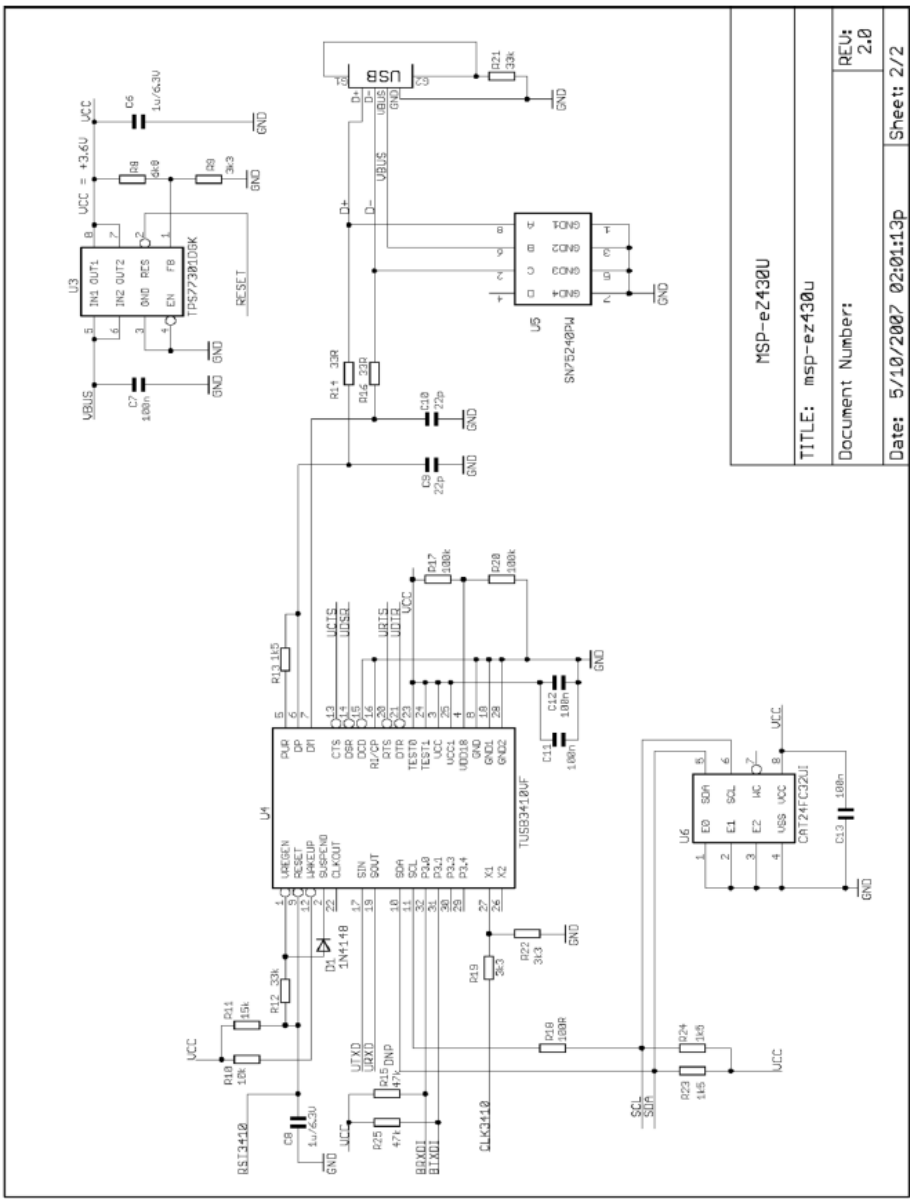


Figura C.2: Diagrama esquemático del ez430-RF2500 USB (cont.).

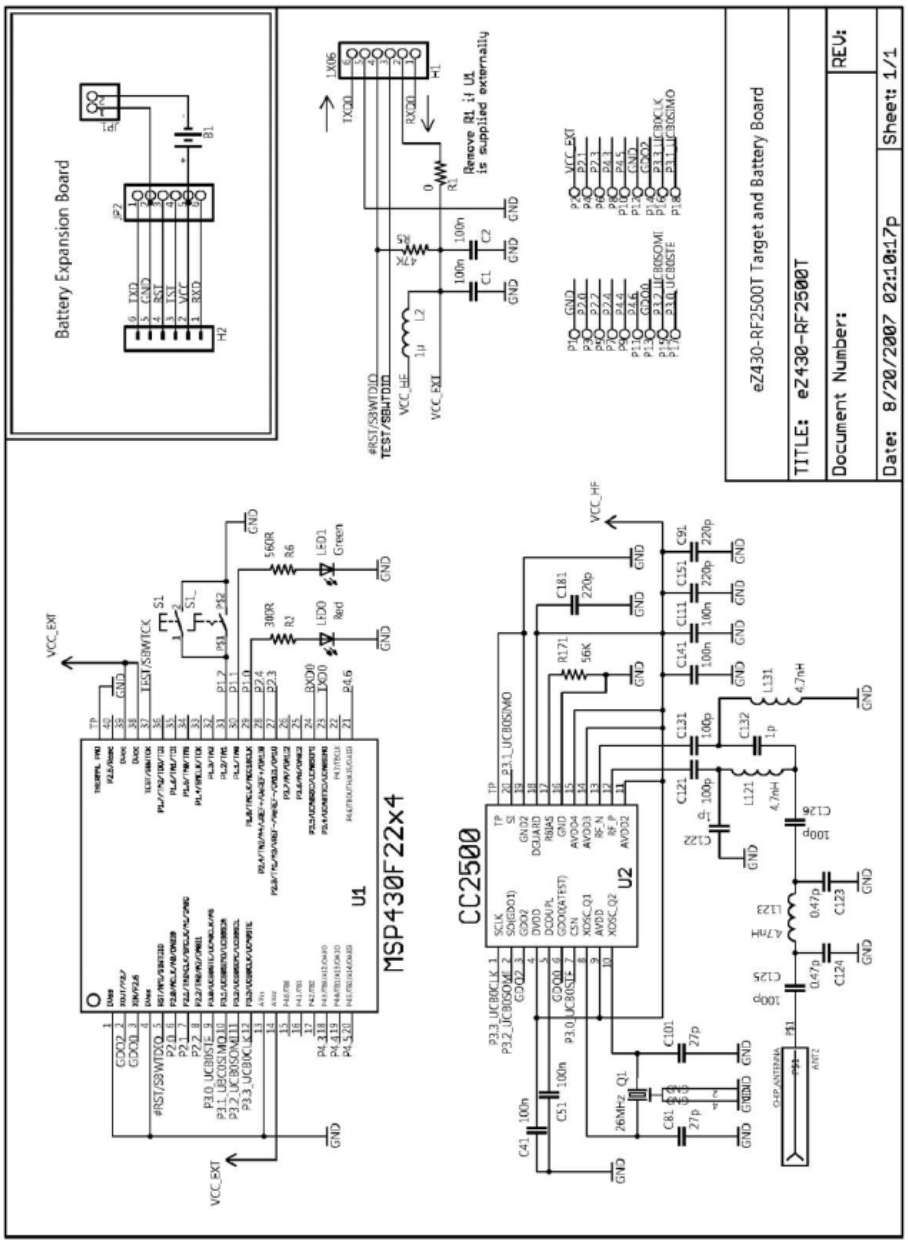


Figura C.3: Diagrama esquemático del ez430-RF2500 Target Board.

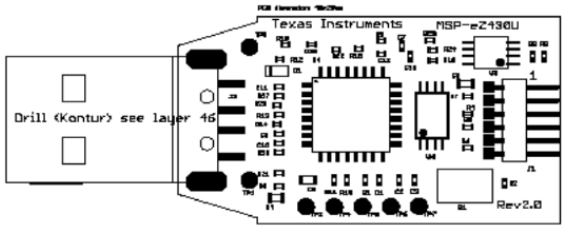


Figure 11. eZ430-RF, USB Debugger, PCB Components Layout

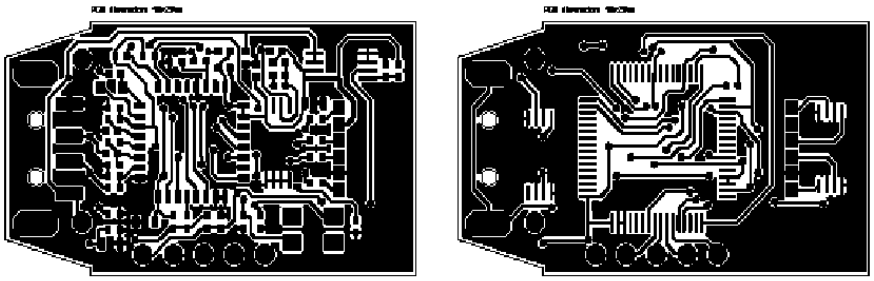


Figure 12. eZ430-RF, USB Debugger, PCB Layout

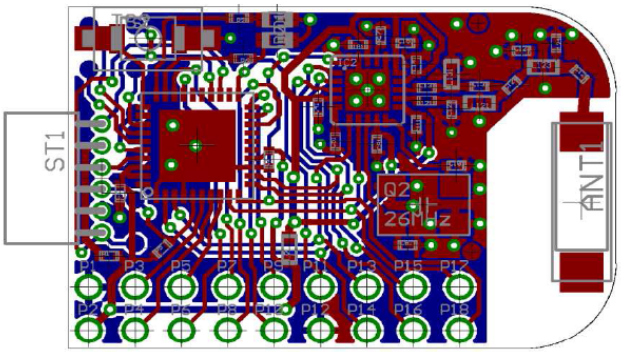


Figura C.4: PCBD del ez430-RF2500.

Apéndice D

WSN Console App, Código Fuente

D.1. Formulario Principal

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Windows.Forms;
9  using System.Drawing.Drawing2D;
10 using WSN.ConsoleClient;
11 using WSN.ConsoleBL;
12 using WSNConsoleClient.Properties;
13
14 namespace WSNConsoleClient
15 {
16     public partial class WSNForm : Form
17     {
18         #region "Private Members"
19
20         private readonly WSNParser _parser = new WSNParser();
21         private List<byte> _serialBuffer = new List<byte>();
22         private static bool _ledState = false;
23         private Timer _timer = new Timer();
24
25         delegate void SetTextCallback(string text);
26
27         private static int _backgroundIndex = 0;
28         private List<Image> _backgroundList = new List<Image>();
29
30         private static int _timerCounterWireless = 0;
31         private static int _timerCounterUSB = 0;
32         private static int _timerWireless = 3;
33         private static int _timerUSB = 3;
34
35         #endregion
36
37         #region "Constructors"
38
39         public WSNForm()
40         {
41             InitializeComponent();
42
43             // Initializes communication manager
44             CommunicationManager.Instance.Parity = "None";
45             CommunicationManager.Instance.StopBits = "1";
46             CommunicationManager.Instance.DataBits = "8";
47             CommunicationManager.Instance.BaudRate = "9600";
48             CommunicationManager.Instance.PortName = "COM32";
49             CommunicationManager.Instance.ReadBufferSize = 1000;
50             CommunicationManager.Instance.CurrentTransmissionType = CommunicationManager.
    TransmissionType.Text;
51             CommunicationManager.Instance.SerialDataReceived += SerialDataReceivedEventHandler;
52
53             // Initializes the graphic's layer
54             AppLayer.Instance.Initialize(DrawingPanel, 120, 140, 350);
55
56             // Load background iamges
57             _backgroundList.Add((Image)Resources.Background);
58             _backgroundList.Add((Image)Resources.Background2);
59             _backgroundList.Add((Image)Resources.Background3);
60             _backgroundList.Add((Image)Resources.Background4);
61             _backgroundList.Add((Image)Resources.Background5);
62             _backgroundList.Add((Image)Resources.Background6);
63             _backgroundList.Add((Image)Resources.Background7);
64
65             _timer.Interval = 1000;
66             _timer.Tick += new EventHandler(_timer_Tick);
67             _timer.Start();
68
69             TimerEffects.Enabled = true;
70         }
71
```

```
72          #endregion
73
74          #region "Form Events"
75
76          private void WSNForm_Load(object sender, EventArgs e)
77          {
78              try
79              {
80                  CommunicationManager.Instance.OpenPort();
81              }
82              catch (Exception ex)
83              {
84                  MessageBox.Show(ex.Message);
85                  Application.Exit();
86              }
87          }
88
89          private void WSNForm_FormClosed(object sender, FormClosedEventArgs e)
90          {
91              try
92              {
93                  CommunicationManager.Instance.ClosePort();
94                  Application.Exit();
95              }
96              catch
97              {}
98          }
99
100         private void SerialDataReceivedEventHandler(object sender, System.IO.Ports.
    SerialDataReceivedEventArgs e)
101         {
102             lock (this)
103             {
104                 try
105                 {
106                     var inputData = CommunicationManager.Instance.ReadExisting();
107                     _parser.AddData(inputData);
108                 }
109                 catch (Exception exception)
110                 {
111                     Console.WriteLine(exception);
112                 }
113             }
114         }
115
116         private void _timer_Tick(object sender, EventArgs e)
117         {
118             try
119             {
120                 if (_parser.BufferCount > 100)
121                 {
122                     var parsedData = _parser.Parse();
123
124                     WSNManager.Instance.UpdateWSNStatus(parsedData);
125                 }
126
127                 var nodesStatus = WSNManager.Instance.GetAllNodesStatus();
128
129                 if (nodesStatus == null)
130                 {
131                     return;
132                 }
133
134                 foreach (var nodeStatus in nodesStatus)
135                 {
136                     var textData = nodeStatus.ToString() + Environment.NewLine;
137                     SetText(textData);
138                 }
139
140                 if (TextBoxData.Lines.ToList().Count > 80)
141                 {
142                     TextBoxData.Clear();
```

```
143                 }
144
145                 AppLayer.Instance.UpdateWSNStatus();
146                 AppLayer.Instance.DrawWSNStatus(DrawingPanel, _backgroundList[_backgroundIndex]);
147             }
148             catch
149             { }
150         }
151
152         private void DrawingPanel_Paint(object sender, PaintEventArgs e)
153         {
154             /*Draw the _graphicsImage onto the DrawingPanel. In this, instead
155               of using DrawingPanel.CreateGraphics(), we can use e.Graphics instead.
156               It will return the graphics used to paint DrawingPanel. This will do the
157               same as DrawingPanel.CreateGraphics(), but i thought i should point
158               this out */
159             e.Graphics.DrawImageUnscaled(AppLayer.Instance.GraphicImage, new Point(0, 0));
160         }
161
162         private void ClearButton_Click(object sender, EventArgs e)
163         {
164             AppLayer.Instance.ClearPanel(DrawingPanel);
165             TextBoxData.Clear();
166         }
167
168         private void RedrawButton_Click(object sender, EventArgs e)
169         {
170             _ledState = !_ledState;
171
172             var cmdOn = (0x49).ToString();  // 01001001
173             var cmdOff = 0x41.ToString();   // 01000001
174
175             CommunicationManager.Instance.WriteData(_ledState ? cmdOn : cmdOff);
176
177             //AppLayer.Instance.UpdateWSNStatus();
178             //AppLayer.Instance.DrawWSNStatus(DrawingPanel, _backgroundList[_backgroundIndex]);
179         }
180
181         private void TimerEffects_Tick(object sender, EventArgs e)
182         {
183             if (_timerCounterWireless++ == _timerWireless)
184             {
185                 PictureBoxMonitor.Visible = !PictureBoxMonitor.Visible;
186
187                 var rnd = new Random(DateTime.Now.Millisecond);
188                 _timerWireless = rnd.Next(1, 7);
189                 _timerCounterWireless = 0;
190             }
191
192             if (_timerCounterUSB++ == _timerUSB)
193             {
194                 PictureBoxUSB.Visible = !PictureBoxUSB.Visible;
195
196                 var rnd = new Random(DateTime.Now.Millisecond);
197                 _timerUSB = rnd.Next(1, 7);
198                 _timerCounterUSB = 0;
199             }
200         }
201
202         private void DrawingPanel_DoubleClick(object sender, EventArgs e)
203         {
204             if (_backgroundIndex++ >= 6)
205             {
206                 _backgroundIndex = 0;
207             }
208
209         }
210
211         #endregion
212
213         #region "Public Methods"
214
```

```
215         public void StartTimer()
216         {
217             _timer.Start();
218             TimerEffects.Start();
219             WSNManager.Instance.StartTimer();
220         }
221 
222         public void StopTimer()
223         {
224             _timer.Stop();
225             TimerEffects.Stop();
226             WSNManager.Instance.StopTimer();
227         }
228 
229         #endregion
230 
231         #region "Private Methods"
232 
233         // This method demonstrates a pattern for making thread-safe
234         // calls on a Windows Forms control.
235         private void SetText(string text)
236         {
237             // InvokeRequired required compares the thread ID of the
238             // calling thread to the thread ID of the creating thread.
239             // If these threads are different, it returns true.
240             if (this.TextBoxData.InvokeRequired)
241             {
242                 SetTextCallback d = new SetTextCallback(SetText);
243                 this.Invoke(d, new object[] { text });
244             }
245             else
246             {
247                 this.TextBoxData.Text += text;
248             }
249         }
250 
251         #endregion
252     }
253 }
254 
```

D.2. Capa de Aplicación

```
1  // ------------------------------------------------------------------------------------------------------
2  // <copyright file="AppLayer.cs" company="TechBits">
3  //   Leandro G. Vacirca.
4  // </copyright>
5  // <summary>
6  //   Defines the AppLayer type.
7  // </summary>
8  // ------------------------------------------------------------------------------------------------------
9
10 namespace WSN.ConsoleClient
11 {
12     using System;
13     using System.Collections.Generic;
14     using System.Linq;
15     using System.Drawing;
16     using WSN.ConsoleBL;
17     using System.Windows.Forms;
18     using System.Drawing.Drawing2D;
19     using WSNConsoleClient.Properties;
20
21     /// <summary>
22     /// Application Layer class.
23     /// </summary>
24     public class AppLayer
25     {
26         #region "Private Members"
27
28         /// <summary>
29         /// Unique instance of the WSNCommandsManager class.
30         /// </summary>
31         private static readonly AppLayer _instance = new AppLayer();
32
33         /// <summary>
34         /// The graphics object we use will draw on this image. This image will
35         /// then be drawn on the drawing surface (DrawingPanel). If we drew directly
36         /// on the DrawingPanel, there would be no way to save what we drew.
37         /// </summary>
38         private Bitmap _graphicsImage;
39
40         /// <summary>
41         /// List of nodes statuses.
42         /// </summary>
43         private Dictionary<string, GraphWSNNode> _wsnGraphNodes = new Dictionary<string,
       GraphWSNNode>();
44
45         private const int BaseAngle = 45;
46
47         #endregion
48
49         #region "Constructors"
50
51         /// <summary>
52         /// Initializes static members of the <see cref="AppLayer"/> class.
53         /// </summary>
54         /// <remarks>
55         /// Explicit static constructor to tell C# compiler not to mark type as beforefieldinit.
56         /// </remarks>
57         static AppLayer() {}
58
59         /// <summary>
60         /// Prevents a default instance of the <see cref="AppLayer"/> class from being created.
61         /// </summary>
62         private AppLayer() {}
63
64         #endregion
65
66         #region "Public Properties"
67
68         /// <summary>
69         /// Gets the unique instance of the AppLayer class.
```

```
70          /// </summary>
71          public static AppLayer Instance
72          {
73              get { return _instance; }
74          }
75
76          public Bitmap GraphicImage
77          {
78              get
79              {
80                  return _graphicsImage;
81              }
82              set
83              {
84                  _graphicsImage = value;
85              }
86          }
87
88          public int CircleRadiousEndDevice { get; private set; }
89
90          public int CircleRadiousAP { get; private set; }
91
92          public int WindowRadious { get; private set; }
93
94          public Point PointCenter { get; private set; }
95
96          #endregion
97
98          #region "Public Methods"
99
100         public void Initialize(Panel panel, int circleRadiousEndDevice, int circleRadiousAP, int
    windowRadious)
101         {
102             // At this point, _graphicsImage is null. This will assign it a value of a new image
    that is the size of the DrawingPanel
103             _graphicsImage = new Bitmap(panel.Width, panel.Height, System.Drawing.Imaging.
    PixelFormat.Format24bppRgb);
104
105             // Fills the image we just created with white
106             Graphics.FromImage(_graphicsImage).Clear(Color.White);
107
108             CircleRadiousEndDevice = circleRadiousEndDevice;
109             CircleRadiousAP = circleRadiousAP;
110             WindowRadious = windowRadious;
111
112             PointCenter = new Point {
113                 X = (panel.Size.Width / 2) - panel.Location.X - circleRadiousAP / 2,
114                 Y = (panel.Size.Height / 2) - panel.Location.Y - circleRadiousAP / 2 };
115
116             WSNManager.Instance.OnWSNNodeRemoved += new WSNManager.WSNEvent
    (WSNManager_OnWSNNodeRemoved);
117         }
118
119         public void UpdateWSNStatus()
120         {
121             lock (this)
122             {
123                 var nodesStatus = WSNManager.Instance.GetAllNodesStatus();
124                 if (nodesStatus == null)
125                 {
126                     return;
127                 }
128
129                 CheckConsistency(nodesStatus);
130
131                 foreach (var node in nodesStatus)
132                 {
133                     UpdateGraphNode(node);
134                 }
135             }
136         }
137
```

```
138         private void CheckConsistency(List<WSNNodeStatus> wsnNodesStatus)
139         {
140             lock (this)
141             {
142                 var wsnNodeKeys = wsnNodesStatus.Select(n => n.Id);
143                 var nodesToBeRemoved = _wsnGraphNodes.Keys.SkipWhile(n => wsnNodeKeys.Contains(n)). ↙
        ToList();
144 
145                 if (nodesToBeRemoved.Count > 0)
146                 {
147                     foreach (var node in nodesToBeRemoved)
148                     {
149                         _wsnGraphNodes.Remove(node);
150                     }
151                 }
152             }
153         }
154 
155         public void DrawWSNStatus(Panel panel, Image background)
156         {
157             lock (this)
158             {
159                 try
160                 {
161                     var drawingGraphics = Graphics.FromImage(_graphicsImage);
162 
163                     // Set drawing graphics properties
164                     drawingGraphics.CompositingQuality = CompositingQuality.HighQuality;
165                     drawingGraphics.InterpolationMode = InterpolationMode.HighQualityBilinear;
166                     drawingGraphics.SmoothingMode = SmoothingMode.HighQuality;
167 
168                     // Clear panel
169                     drawingGraphics.Clear(Color.White);
170                     drawingGraphics.DrawImage(background, new Rectangle(new Point(0, 0), new Size   ↙
        (panel.Width, panel.Height)));
171 
172                     IEnumerable<string> query = _wsnGraphNodes.Keys.OrderBy(n => n);
173                     foreach (var node in query)
174                     {
175                         _wsnGraphNodes[node].Blink = !_wsnGraphNodes[node].Blink;
176                         DrawNode(node, true);
177                     }
178 
179                     // Draw the _graphicsImage onto the DrawingPanel
180                     panel.CreateGraphics().DrawImageUnscaled(_graphicsImage, new Point(0, 0));
181                 }
182                 catch
183                 {
184                 }
185             }
186         }
187 
188         public void ClearPanel(Panel panel)
189         {
190             // Fills the image with white
191             Graphics.FromImage(_graphicsImage).Clear(Color.White);
192 
193             // Draw the _graphicsImage onto the DrawingPanel
194             panel.CreateGraphics().DrawImageUnscaled(_graphicsImage, new Point(0, 0));
195         }
196 
197         #endregion
198 
199         #region "Private Methods"
200 
201         private void UpdateGraphNode(WSNNodeStatus wsnNodeStatus)
202         {
203             if (!_wsnGraphNodes.ContainsKey(wsnNodeStatus.Id))
204             {
205                 var newNode = new GraphWSNNode();
206 
207                 // Set the node's angle
```

```
208                 newNode.Angle = (short)(GetRandomAngle() * BaseAngle);
209
210                 if (wsnNodeStatus.Id.Contains("HUB"))
211                 {
212                     // If the node is the AP, it will be centered on the screen
213                     newNode.PointCenter = PointCenter;
214                     newNode.PointAdjacent = PointCenter;
215                     newNode.Radious = CircleRadiousAP;
216                 }
217                 else
218                 {
219                     var rssiCorrection = (int)Math.Round((double)WindowRadious * (45 - Convert.
    ToInt32(wsnNodeStatus.RSSI) + 10) / 45, 0);
220
221                     newNode.PointCenter = new Point(
222                         PointCenter.X + Convert.ToInt32(rssiCorrection * Math.Cos(newNode.Angle * 2
    * Math.PI / 360)),
223                         PointCenter.Y + Convert.ToInt32(rssiCorrection * Math.Sin(newNode.Angle * 2
    * Math.PI / 360)));
224
225                     newNode.PointAdjacent = new Point(PointCenter.X, PointCenter.Y);
226                     newNode.Radious = CircleRadiousEndDevice;
227                 }
228
229                 // Update node's data
230                 newNode.Update(wsnNodeStatus);
231
232                 // Set node's style
233                 SetNodeStyle(newNode);
234
235                 // Add node to dictionary
236                 _wsnGraphNodes.Add(wsnNodeStatus.Id, newNode);
237             }
238             else
239             {
240                 if (!wsnNodeStatus.Id.Contains("HUB"))
241                 {
242                     var node = _wsnGraphNodes[wsnNodeStatus.Id];
243                     var rssiCorrection = (int)Math.Round((double)WindowRadious * (45 - Convert.
    ToInt32(node.RSSI) + 10) / 45, 0);
244
245                     node.PointCenter = new Point(
246                         PointCenter.X + Convert.ToInt32(rssiCorrection * Math.Cos(node.Angle * 2 *
    Math.PI / 360)),
247                         PointCenter.Y + Convert.ToInt32(rssiCorrection * Math.Sin(node.Angle * 2 *
    Math.PI / 360)));
248
249                     SetNodeStyle(node);
250                 }
251
252                 // Update node's status
253                 _wsnGraphNodes[wsnNodeStatus.Id].Update(wsnNodeStatus);
254             }
255         }
256
257         private void DrawNode(string nodeID, bool blink)
258         {
259             var node = _wsnGraphNodes[nodeID];
260             if (node == null) return;
261
262             var rect = new Rectangle(node.PointCenter.X, node.PointCenter.Y, node.Radious, node.
    Radious);
263             LinearGradientBrush gradientBrush = node.Blink ? node.BrushBlink : node.BrushDefault;
264             Color borderColor = node.Blink ? node.BorderColorBlink : node.BorderColorDefault;
265
266             if (gradientBrush != null && borderColor != null)
267             {
268                 var graphics = Graphics.FromImage(_graphicsImage);
269
270                 if (!node.EnergyType.Equals(WSN.ConsoleBL.WSNNodeStatus.EnergyTypeEnum.USB))
271                 {
272                     DrawLine(graphics,
```

```
273                         new Point(node.PointCenter.X + (node.Radious / 2), node.PointCenter.Y +
    (node.Radious / 2)),
274                         new Point(node.PointAdjacent.X + (CircleRadiousAP / 2), node.PointAdjacent.Y
     + (CircleRadiousAP / 2)), Color.LightSlateGray);
275                 }
276
277                 // Draw primitives
278                 if (node.EnergySource || node.Id.Contains("HUB"))
279                 {
280                     DrawCircle(graphics, rect.X, rect.Y, node.Radious, gradientBrush, node.Blink ? 4
     : 2, borderColor);
281                 }
282                 else
283                 {
284                     DrawSquare(graphics, rect.X, rect.Y, node.Radious, gradientBrush, node.Blink ? 4
     : 2, borderColor);
285                 }
286
287                 // Draw node's data
288                 DrawNodeData(graphics, node);
289             }
290         }
291
292         private void DrawCircle(Graphics graphics, int x, int y, int radious, LinearGradientBrush
    brush, int edgeWidth, Color edgeColor)
293         {
294             // Draw the filled circle
295             graphics.FillEllipse(brush, new Rectangle(x, y, radious, radious));
296
297             // Create the circle's edge line
298             graphics.DrawEllipse(new Pen(edgeColor, edgeWidth), new Rectangle(x - 1, y - 1, radious
    + 1, radious + 1));
299         }
300
301         private void DrawSquare(Graphics graphics, int x, int y, int radious, LinearGradientBrush
    brush, int edgeWidth, Color edgeColor)
302         {
303             // Draw the filled rectangle
304             graphics.FillRectangle(brush, new Rectangle(x, y, radious, radious));
305
306             // Create the rectangle's edge line
307             graphics.DrawRectangle(new Pen(edgeColor, edgeWidth), new Rectangle(x - 1, y - 1,
    radious + 1, radious + 1));
308         }
309
310         private void DrawString(Graphics graphics, String message, float x, float y, float radious,
    Font fontType, Color fontColor)
311         {
312             // Set format of string
313             var drawFormat = new StringFormat();
314             drawFormat.Alignment = StringAlignment.Center;
315             drawFormat.LineAlignment = StringAlignment.Center;
316
317             SolidBrush drawBrush = new SolidBrush(fontColor);
318
319             // Create point for upper-left corner of drawing
320             var drawRect = new RectangleF(x, y, radious, radious);
321
322             // Draw string to screen
323             graphics.DrawString(message, fontType, drawBrush, drawRect, drawFormat);
324         }
325
326         private void DrawLine(Graphics graphics, Point p1, Point p2, Color lineColor)
327         {
328             // AntiAliasing is to be used
329             graphics.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
330
331             // Draw the line
332             graphics.DrawLine(new Pen(lineColor, 2), p1, p2);
333         }
334
335         private void DrawNodeData(Graphics graphics, GraphWSNNode node)
```

```
336         {
337             // Create string and font to draw temperature value
338             var drawString = node.Temperature.Replace("C", "°C");
339             var drawFont = new Font("Calibri", 12, FontStyle.Bold);
340 
341             DrawString(graphics, drawString, node.PointCenter.X, node.PointCenter.Y - (node.Radious ↙
    * 0.3F), node.Radious, drawFont, Color.White);
342 
343             drawString = node.Id.Equals("HUB0") ? "Access Point" : "Node #" + node.Id.Trim('0') + " ↙
    " + Enum.GetName(typeof(WSNNodeStatus.EnergyTypeEnum), node.EnergyType);
344             drawFont = new Font("Calibri", 10, FontStyle.Bold);
345             DrawString(graphics, drawString, node.PointCenter.X, node.PointCenter.Y, node.Radious,  ↙
    drawFont, Color.Black);
346 
347             // Create string and font to draw the RSSI value
348             var rssi = "RSSI: " + node.RSSI.TrimStart('0') + "%";
349             var moreInfo = node.NumberTx > 0 ? String.Concat(rssi, " Tx: ", node.NumberTx) : rssi;
350 
351             drawString = node.Id.Equals("HUB0") ? "" : moreInfo;
352             drawFont = new Font("Arial", 7, FontStyle.Bold);
353             DrawString(graphics, drawString, node.PointCenter.X, node.PointCenter.Y + (node.Radious ↙
    * 0.13F), node.Radious, drawFont, Color.Blue);
354 
355             // Create string and font to draw voltage value
356             drawString = string.Concat(node.Voltage.ToString(), "V");
357             drawFont = new Font("Calibri", 12, FontStyle.Bold);
358             DrawString(graphics, drawString, node.PointCenter.X, node.PointCenter.Y + (node.Radious ↙
    * 0.3F), node.Radious, drawFont, Color.SaddleBrown);
359 
360             if (node.ButtonPressed)
361             {
362                 // Create the circle's edge line
363                 var rect = new Rectangle(node.PointCenter.X, node.PointCenter.Y, node.Radious, node.↙
    Radious);
364 
365                 var coeff = (int)(node.Radious * 0.16);
366                 graphics.DrawEllipse(new Pen(node.BrushDefault, 5), new Rectangle((int)rect.X -     ↙
    coeff/2, (int)rect.Y - coeff/2, (int)node.Radious + coeff, (int)node.Radious + coeff));
367             }
368         }
369 
370         private void SetNodeStyle(GraphWSNNode wsnNode)
371         {
372             var rect = new Rectangle(wsnNode.PointCenter.X, wsnNode.PointCenter.Y, wsnNode.Radious, ↙
    wsnNode.Radious);
373 
374             switch (wsnNode.EnergyType)
375             {
376                 case (int)WSNNodeStatus.EnergyTypeEnum.USB:
377                     {
378                         wsnNode.BrushDefault = new LinearGradientBrush(rect, Color.OrangeRed, Color.↙
    WhiteSmoke, LinearGradientMode.ForwardDiagonal);
379                         wsnNode.BrushBlink = new LinearGradientBrush(rect, Color.OrangeRed, Color.  ↙
    IndianRed, LinearGradientMode.ForwardDiagonal);
380                         wsnNode.BorderColorDefault = Color.IndianRed;
381                         wsnNode.BorderColorBlink = Color.DarkRed;
382 
383                         break;
384                     }
385 
386                 case (int)WSNNodeStatus.EnergyTypeEnum.Battery:
387                     {
388                         wsnNode.BrushDefault = new LinearGradientBrush(rect, Color.LawnGreen, Color.↙
    WhiteSmoke, LinearGradientMode.ForwardDiagonal);
389                         wsnNode.BrushBlink = new LinearGradientBrush(rect, Color.LawnGreen, Color.  ↙
    LightGreen, LinearGradientMode.ForwardDiagonal);
390                         wsnNode.BorderColorDefault = Color.ForestGreen;
391                         wsnNode.BorderColorBlink = Color.DarkGreen;
392 
393                         break;
394                     }
395 
```

```
396                 case (int)WSNNodeStatus.EnergyTypeEnum.RF:
397                     {
398                         wsnNode.BrushDefault = new LinearGradientBrush(rect, Color.Aquamarine, Color
    .White, LinearGradientMode.ForwardDiagonal);
399                         wsnNode.BrushBlink = new LinearGradientBrush(rect, Color.Aquamarine, Color.
    LightBlue, LinearGradientMode.ForwardDiagonal);
400                         wsnNode.BorderColorDefault = Color.LightBlue;
401                         wsnNode.BorderColorBlink = Color.CadetBlue;
402 
403                         break;
404                     }
405 
406                 case (int)WSNNodeStatus.EnergyTypeEnum.Solar:
407                     {
408                         wsnNode.BrushDefault = new LinearGradientBrush(rect, Color.Gold, Color.
    GhostWhite, LinearGradientMode.ForwardDiagonal);
409                         wsnNode.BrushBlink = new LinearGradientBrush(rect, Color.Gold, Color.
    LightGoldenrodYellow, LinearGradientMode.ForwardDiagonal);
410                         wsnNode.BorderColorDefault = Color.Goldenrod;
411                         wsnNode.BorderColorBlink = Color.DarkGoldenrod;
412 
413                         break;
414                     }
415             }
416         }
417 
418         private int GetRandomAngle()
419         {
420             var nodeAngles = _wsnGraphNodes.Values.Select(n => n.Angle / BaseAngle).ToList();
421 
422             int angle = 0;
423             var randomAngle = new Random();
424 
425             var count = (360 / BaseAngle) - 1;
426             for (var i = 0; i < count; i++)
427             {
428                 angle = randomAngle.Next(0, count);
429 
430                 if (!nodeAngles.Contains(angle))
431                 {
432                     break;
433                 }
434             }
435 
436             return angle;
437         }
438 
439         private void WSNManager_OnWSNNodeRemoved(string nodeId)
440         {
441         }
442 
443         #endregion
444     }
445 }
```

D.3. Wireless Sensor Network Manager

```
// ------------------------------------------------------------------------------------------------------------
// <copyright file="WSNCommandsManager.cs" company="TechBits">
//   Leandro G. Vacirca.
// </copyright>
// <summary>
//   Defines the WSNCommandsManager type.
// </summary>
// ------------------------------------------------------------------------------------------------------------

namespace WSN.ConsoleBL
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Threading;
    using WSN.ExtensionMethods;

    /// <summary>
    /// WSN Manager class.
    /// </summary>
    public class WSNManager
    {
        #region "Private Members"

        /// <summary>
        /// Unique instance of the WSNCommandsManager class.
        /// </summary>
        private static readonly WSNManager _instance = new WSNManager();

        /// <summary>
        /// List of nodes statuses.
        /// </summary>
        private static List<WSNNodeStatus> _wsnNodesStatus = new List<WSNNodeStatus>();

        /// <summary>
        /// Status checker timer.
        /// </summary>
        private Timer _timerStatus;

        /// <summary>
        /// Time -out for removing a node from the list.
        /// </summary>
        private const int TIME_OUT = 4;

        private const int TIME_OUT_HARVESTERS = 12;

        #endregion

        #region "Constructors"

        /// <summary>
        /// Initializes static members of the <see cref="WSNManager"/> class.
        /// </summary>
        /// <remarks>
        /// Explicit static constructor to tell C# compiler not to mark type as beforefieldinit.
        /// </remarks>
        static WSNManager()
        {
        }

        /// <summary>
        /// Prevents a default instance of the <see cref="WSNManager"/> class from being created.
        /// </summary>
        private WSNManager()
        {
            _timerStatus = new Timer(new TimerCallback(WSNStatusChecker), null, 0, 1000);
            OnWSNNodeRemoved += OnWSNNodeRemovedEventHandler;
        }

        #endregion
```

```
71
72          #region "Public Properties"
73
74          /// <summary>
75          /// Gets the unique instance of the ZFlyCylCommandsManager class.
76          /// </summary>
77          public static WSNManager Instance
78          {
79              get { return _instance; }
80          }
81
82          public event WSNEvent OnWSNNodeRemoved;
83          public delegate void WSNEvent(string nodeId);
84
85          #endregion
86
87          #region "Public Methods"
88
89          public bool UpdateWSNStatus(List<WSNNodeStatus> nodesStatus)
90          {
91              if (nodesStatus.Count == 0)
92              {
93                  return false;
94              }
95
96              var uniqueNodes = nodesStatus.Distinct().ToList();
97
98              foreach (var node in uniqueNodes)
99              {
100                 UpdateNodeStatus(node);
101             }
102
103            return true;
104         }
105
106         /// <summary>
107         /// Returns the all nodes status.
108         /// </summary>
109         /// <returns>The status of all nodes.</returns>
110         public List<WSNNodeStatus> GetAllNodesStatus()
111         {
112             lock (this)
113             {
114                 if (_wsnNodesStatus.Count == 0)
115                 {
116                     return null;
117                 }
118
119                 return _wsnNodesStatus;
120             }
121         }
122
123         /// <summary>
124         /// Defines the temperature display option.
125         /// </summary>
126         /// <param name="temperatureMode">
127         /// C - Output all temperatures in degrees Celsius.
128         /// F - Output all temperatures in degrees Fahrenheit.
129         /// </param>
130         public void TemperatureMode(char temperatureMode)
131         {
132             CommunicationManager.Instance.WriteData(temperatureMode.ToString());
133         }
134
135         /// <summary>
136         /// Defines the data format option.
137         /// </summary>
138         /// <param name="dataFormatMode">
139         /// V - Output all data in extended Verbose mode.
140         /// Node:HUB0,Temp: 91.2F,Baterry:3.5V,Strength:000%,ButtonPressed:no
141         /// M - Output all data in shortened Minimal mode.
142         /// $HUB0,89.6F,3.5,000,N#
```

```
143         /// </param>
144         public void DataFormatMode(char dataFormatMode)
145         {
146             CommunicationManager.Instance.WriteData(dataFormatMode.ToString());
147         }
148 
149         public void StartTimer()
150         {
151             _timerStatus = new Timer(new TimerCallback(WSNStatusChecker), null, 0, 1000);
152         }
153 
154         public void StopTimer()
155         {
156             _timerStatus = new Timer(new TimerCallback(WSNStatusChecker), null, Timeout.Infinite,
    1000);
157         }
158 
159         #endregion
160 
161         #region "Private Methods"
162         /// <summary>
163         /// Updates the node status.
164         /// </summary>
165         /// <param name="nodeStatus">The new node status.</param>
166         private void UpdateNodeStatus(WSNNodeStatus nodeStatus)
167         {
168             lock (this)
169             {
170                 var node = new WSNNodeStatus();
171                 if (nodeStatus == null)
172                 {
173                     return;
174                 }
175 
176                 try
177                 {
178                     // Find the node in the internal list
179                     node = _wsnNodesStatus.First(n => n.Equals(nodeStatus));
180 
181                     // Update node status
182                     node.Update(nodeStatus);
183                 }
184                 catch (InvalidOperationException)
185                 {
186                     // Add node in case it does not exist
187                     var newNode = nodeStatus.Clone<WSNNodeStatus>();
188                     newNode.Update(nodeStatus);
189 
190                     _wsnNodesStatus.Add(newNode);
191                 }
192             }
193         }
194 
195         private void WSNStatusChecker(object obj)
196         {
197             lock (this)
198             {
199                 var nodesToRemove = new List<string>();
200 
201                 // Check nodes to be removed
202                 foreach (var node in _wsnNodesStatus)
203                 {
204                     if (!node.EnergyType.Equals((int)WSNNodeStatus.EnergyTypeEnum.USB))
205                     {
206                         if (node.EnergyType.Equals((int)WSNNodeStatus.EnergyTypeEnum.Solar) || node.
    EnergyType.Equals((int)WSNNodeStatus.EnergyTypeEnum.RF))
207                         {
208                             if (DateTime.Now.Subtract(node.LastUpdate).TotalSeconds >
    TIME_OUT_HARVESTERS)
209                             {
210                                 nodesToRemove.Add(node.Id);
211                             }
```

```
212                         }
213                         else if (DateTime.Now.Subtract(node.LastUpdate).TotalSeconds > TIME_OUT)
214                         {
215                             nodesToRemove.Add(node.Id);
216                         }
217                     }
218                 }
219 
220                 if (nodesToRemove.Count == 0)
221                 {
222                     return;
223                 }
224 
225                 // Remove the nodes
226                 foreach (var node in nodesToRemove)
227                 {
228                     var nodeToRemove = (WSNNodeStatus)_wsnNodesStatus.Where(n => n.Id.Equals(node)).↙
    First();
229                     _wsnNodesStatus.Remove(nodeToRemove);
230                 }
231             }
232         }
233 
234         /// <summary>Event that occurs when a node is removed.</summary>
235         private void OnWSNNodeRemovedEventHandler(string nodeId)
236         {
237             if (OnWSNNodeRemoved != null)
238             {
239                 OnWSNNodeRemoved(nodeId);
240             }
241         }
242 
243         #endregion
244     }
245 }
```

D.4. Gestor de Comunicación Serial

```
1  // ------------------------------------------------------------------------------------------------------
2  // <copyright file="CommunicationManager.cs" company="TechBits">
3  //   Leandro G. Vacirca.
4  // </copyright>
5  // <summary>
6  //   Defines the CommunicationManager type.
7  // </summary>
8  // ------------------------------------------------------------------------------------------------------
9
10 namespace WSN.ConsoleBL
11 {
12     using System;
13     using System.Collections.Generic;
14     using System.IO.Ports;
15     using System.Text;
16     using System.Linq;
17
18     /// <summary>
19     /// Manager para las comunicaciones via puerto serie.
20     /// </summary>
21     public class CommunicationManager
22     {
23         #region "Private Members"
24
25         /// <summary>
26         /// Unique instance of the CommunicationManager class.
27         /// </summary>
28         private static readonly CommunicationManager _instance = new CommunicationManager();
29
30         /// <summary>
31         /// Comm port.
32         /// </summary>
33         private readonly SerialPort _comPort = new SerialPort();
34
35         #endregion
36
37         #region "Constructors"
38
39         /// <summary>
40         /// Initializes static members of the <see cref="CommunicationManager"/> class.
41         /// </summary>
42         /// <remarks>
43         /// Explicit static constructor to tell C# compiler not to mark type as beforefieldinit.
44         /// </remarks>
45         static CommunicationManager()
46         {
47         }
48
49         /// <summary>
50         /// Prevents a default instance of the <see cref="CommunicationManager"/> class from being
       created.
51         /// </summary>
52         private CommunicationManager()
53         {
54             BaudRate = string.Empty;
55             Parity = string.Empty;
56             StopBits = string.Empty;
57             DataBits = string.Empty;
58             PortName = "COM1";
59
60             _comPort.DataReceived += ComPortDataReceived;
61         }
62
63         #endregion
64
65         #region "Events"
66
67         public event SerialDataReceivedEventHandler SerialDataReceived;
68
69         #endregion
```

```
70
71          #region "Enums"
72
73          public enum TransmissionType
74          {
75              Text,
76              Hex
77          }
78
79          #endregion
80
81          #region "Public Properties"
82
83          /// <summary>
84          /// Gets the unique instance of the CommunicationManager class.
85          /// </summary>
86          public static CommunicationManager Instance
87          {
88              get { return _instance; }
89          }
90
91          /// <summary>
92          /// Gets or sets BaudRate.
93          /// </summary>
94          public string BaudRate { get; set; }
95
96                  /// <summary>
97          /// Gets or sets Parity.
98          /// </summary>
99          public string Parity { get; set; }
100
101         /// <summary>
102         /// Gets or sets StopBits.
103         /// </summary>
104         public string StopBits { get; set; }
105
106         /// <summary>
107         /// Gets or sets DataBits.
108         /// </summary>
109         public string DataBits { get; set; }
110
111         /// <summary>
112         /// Gets or sets PortName.
113         /// </summary>
114         public string PortName { get; set; }
115
116         /// <summary>
117         /// Gets or sets ReadBufferSize.
118         /// </summary>
119         public int ReadBufferSize { get; set; }
120
121         /// <summary>
122         /// Gets BytesToRead.
123         /// </summary>
124         public int BytesToRead
125         {
126             get
127             {
128                 return _comPort.BytesToRead;
129             }
130         }
131
132         /// <summary>
133         /// Gets or sets CurrentTransmissionType.
134         /// </summary>
135         public TransmissionType CurrentTransmissionType { get; set; }
136
137         #endregion
138
139         #region "Public Methods"
140
141         public void WriteData(string msg)
```

```
142         {
143             switch (CurrentTransmissionType)
144             {
145                 case TransmissionType.Text:
146                     {
147                         if (!_comPort.IsOpen)
148                         {
149                             _comPort.Open();
150                         }
151 
152                         _comPort.Write(msg);
153 
154                         break;
155                     }
156 
157                 case TransmissionType.Hex:
158                     try
159                     {
160                         byte[] newMsg = HexToByte(msg);
161 
162                         _comPort.Write(newMsg, 0, newMsg.Length);
163                     }
164                     catch (FormatException)
165                     {
166                     }
167 
168                     break;
169 
170                 default:
171 
172                     if (!_comPort.IsOpen)
173                     {
174                         _comPort.Open();
175                     }
176 
177                     _comPort.Write(msg);
178 
179                     break;
180             }
181         }
182 
183         public byte[] ReadData(int bytesCount)
184         {
185             int bytes = _comPort.BytesToRead;
186 
187             if (bytes < bytesCount)
188             {
189                 return null;
190             }
191 
192             var comBuffer = new byte[bytes];
193             _comPort.Read(comBuffer, 0, bytes);
194 
195             return comBuffer;
196         }
197 
198         public byte[] ReadExisting()
199         {
200             return _comPort.ReadExisting().ToCharArray().Select(b => (byte)b).ToArray<byte>();
201         }
202 
203         public bool OpenPort()
204         {
205             try
206             {
207                 if (_comPort.IsOpen)
208                 {
209                     _comPort.Close();
210                 }
211 
212                 _comPort.ReadBufferSize = ReadBufferSize;
213                 _comPort.BaudRate = int.Parse(BaudRate);
```

```
214                 _comPort.DataBits = int.Parse(DataBits);
215                 _comPort.StopBits = (StopBits)Enum.Parse(typeof(StopBits), StopBits);
216                 _comPort.Parity = (Parity)Enum.Parse(typeof(Parity), Parity);
217                 _comPort.PortName = PortName;
218                 _comPort.Handshake = Handshake.None;
219 
220                 _comPort.Open();
221                 return true;
222             }
223             catch (Exception ex)
224             {
225                 throw new Exception(ex.Message);
226             }
227         }
228 
229         public void ClosePort()
230         {
231             try
232             {
233                 if (_comPort.IsOpen)
234                 {
235                     _comPort.Close();
236                 }
237             }
238             catch (Exception)
239             {
240             }
241         }
242 
243         public static byte[] HexToByte(string msg)
244         {
245             msg = msg.Replace(" ", string.Empty);
246             var comBuffer = new byte[msg.Length / 2];
247 
248             for (int i = 0; i < msg.Length; i += 2)
249             {
250                 comBuffer[i / 2] = Convert.ToByte(msg.Substring(i, 2), 16);
251             }
252 
253             return comBuffer;
254         }
255 
256         public static string ByteToHex(ICollection<byte> comByte)
257         {
258             var builder = new StringBuilder(comByte.Count * 3);
259 
260             foreach (byte data in comByte)
261             {
262                 builder.Append(Convert.ToString(data, 16).PadLeft(2, '0').PadRight(3, ' '));
263             }
264 
265             return builder.ToString().ToUpper();
266         }
267 
268         #endregion
269 
270         #region "Private Methods"
271 
272         private void ComPortDataReceived(object sender, SerialDataReceivedEventArgs e)
273         {
274             if (SerialDataReceived != null)
275             {
276                 SerialDataReceived(sender, e);
277             }
278         }
279 
280         #endregion
281     }
282 }
```

D.5. Parser de Datos

```
1  // ------------------------------------------------------------------------------------------------------
2  // <copyright file="WSNParser.cs" company="TechBits">
3  //   Leandro G. Vacirca.
4  // </copyright>
5  // <summary>
6  //   Defines the WSNParser type.
7  // </summary>
8  // ------------------------------------------------------------------------------------------------------
9
10 namespace WSN.ConsoleBL
11 {
12     using System;
13     using System.Collections.Generic;
14     using System.Linq;
15     using System.Text;
16
17     /// <summary>
18     /// Parser for WSN data packets.
19     /// </summary>
20     public class WSNParser
21     {
22         #region "Private Memebers"
23
24         private List<byte> _serialBuffer = new List<byte>();
25
26         #endregion
27
28         #region "Public Methods"
29
30         public void AddData(byte[] serialData)
31         {
32             lock (this)
33             {
34                 if (serialData.Length == 0)
35                 {
36                     return;
37                 }
38
39                 foreach (var ch in serialData)
40                 {
41                     var charByte = (char)ch;
42
43                     if (!charByte.Equals('\r') && !charByte.Equals('\n') && !charByte.Equals('#') &&
       !charByte.Equals('\0'))
44                     {
45                         _serialBuffer.Add(ch);
46                     }
47                 }
48             }
49         }
50
51         public int BufferCount
52         {
53             get
54             {
55                 return _serialBuffer.Count;
56             }
57         }
58
59         /// <summary>
60         /// Performs the parsing algorithms for WSn data packets..
61         /// </summary>
62         public List<WSNNodeStatus> Parse()
63         {
64             lock (this)
65             {
66                 // $ADDR,-XX.XC,V.C,RSI,T,B,XXXXXXX#
67                 var serialBufferArray = _serialBuffer.ToArray();
68
69                 // Parses sentences
```

```
70                 char[] separator = { '$' };
71                 var sentences = Encoding.ASCII.GetString(serialBufferArray).Split(separator,
       StringSplitOptions.RemoveEmptyEntries);
72
73                 // Check if there is any data to process
74                 if (sentences == null || sentences.Length == 0)
75                 {
76                     return null;
77                 }
78
79                 var packets = new List<WSNNodeStatus>();
80                 foreach (var sentence in sentences)
81                 {
82                     if (!String.IsNullOrEmpty(sentence))
83                     {
84                         var parameters = sentence.Split(',');
85
86                         if (!(parameters.Length < 7) && !(parameters[0].Length < 4) && !(parameters
       [6].Length < 7))
87                         {
88                             var id = parameters[0];
89                             var temp = parameters[1];
90                             var volt = parameters[2];
91                             var rssi = parameters[3];
92                             var tes = parameters[4];
93                             var but = parameters[5];
94                             var mode = parameters[6].Substring(0, 1);
95                             var onOff = parameters[6].Substring(3, 1);
96                             var numTx = parameters[6].Substring(4);
97
98                             var newNode = new WSNNodeStatus(id, temp, volt, rssi, but, tes, mode,
       numTx, onOff);
99                             packets.Add(newNode);
100                        }
101                    }
102                }
103
104                _serialBuffer.Clear();
105                return packets;
106            }
107        }
108
109        #endregion
110    }
111 }
```

D.6. El objeto WSN Nodo Gráfico

```
1  // ------------------------------------------------------------------------------------------------------------
       -------------------
2  // <copyright file="GraphWSNNode.cs" company="TechBits">
3  //   Leandro G. Vacirca.
4  // </copyright>
5  // <summary>
6  //   Defines the GraphWSNNode type.
7  // </summary>
8  // ------------------------------------------------------------------------------------------------------------
       -------------------
9
10 namespace WSN.ConsoleBL
11 {
12     using System;
13     using System.Collections.Generic;
14     using System.Drawing;
15 using System.Drawing.Drawing2D;
16
17     /// <summary>
18     /// Graph WSN Node class.
19     /// </summary>
20     public class GraphWSNNode: ICloneable
21     {
22         #region "Constructors"
23
24         /// <summary>
25         /// Initializes a new instance of the <see cref="WSNNodeStatus"/> class.
26         /// </summary>
27         public GraphWSNNode()
28         {
29         }
30
31         #endregion
32
33         #region "Public Properties"
34
35         public string Id { get; set; }
36
37         public bool EnergySource { get; set; }
38
39         public int EnergyType { get; set; }
40
41         public string Temperature { get; set; }
42
43         public string Voltage { get; set; }
44
45         public string RSSI { get; set; }
46
47         public bool ButtonPressed { get; set; }
48
49         public bool Error { get; set; }
50
51         public Point PointCenter { get; set; }
52
53         public Point PointAdjacent { get; set; }
54
55         public int Radious { get; set; }
56
57         public short Angle { get; set; }
58
59         public bool Blink { get; set; }
60
61         public int NumberTx { get; set; }
62
63         public LinearGradientBrush BrushDefault { get; set; }
64
65         public LinearGradientBrush BrushBlink { get; set; }
66
67         public Color BorderColorDefault { get; set; }
68
69         public Color BorderColorBlink { get; set; }
70
```

```
71          #endregion
72
73          #region "Public Methods"
74
75          public void Update(WSNNodeStatus newNodeStatus)
76          {
77              Id = newNodeStatus.Id;
78              Voltage = newNodeStatus.Voltage;
79              Temperature = newNodeStatus.Temperature;
80              RSSI = newNodeStatus.RSSI;
81              ButtonPressed = newNodeStatus.ButtonPressed;
82              EnergySource = newNodeStatus.EnergySource;
83              EnergyType = newNodeStatus.EnergyType;
84              NumberTx = newNodeStatus.NumberTx;
85          }
86
87          /// <summary>
88          /// Prints node status.
89          /// </summary>
90          /// <returns>The node status.</returns>
91          public override string ToString()
92          {
93              return String.Concat('$',
94                  Id, ' ',
95                  Temperature, ' ',
96                  Voltage, ' ',
97                  RSSI, ' ',
98                  Enum.GetName(typeof(WSN.ConsoleBL.WSNNodeStatus.EnergyTypeEnum), EnergyType), '#');
99          }
100
101         public object Clone()
102         {
103             return MemberwiseClone();
104         }
105
106         #endregion
107     }
108 }
109
```

D.7. El objeto WSN Estado de Nodo

```
1  // ------------------------------------------------------------------------------------------------------
2  // <copyright file="WSNPacket.cs" company="TechBits">
3  //   Leandro G. Vacirca.
4  // </copyright>
5  // <summary>
6  //   Defines the WSNPacketFormat type.
7  // </summary>
8  // ------------------------------------------------------------------------------------------------------
9
10 namespace WSN.ConsoleBL
11 {
12     using System;
13     using System.Collections.Generic;
14
15     /// <summary>
16     /// WSN Node Status class.
17     /// </summary>
18     [Serializable]
19     public class WSNNodeStatus : IEquatable<WSNNodeStatus>
20     {
21         #region "Constructors"
22
23         /// <summary>
24         /// Initializes a new instance of the <see cref="WSNNodeStatus"/> class.
25         /// </summary>
26         public WSNNodeStatus()
27         {
28             Bytes = new List<byte>(25);
29         }
30
31         public WSNNodeStatus(string id, string temperature, string voltage, string rssi, string
       buttonPressed, string energyType, string energySource, string numberTx, string onOff)
32         {
33             Id = id;
34             Temperature = temperature;
35             Voltage = voltage;
36             RSSI = rssi;
37             ButtonPressed = buttonPressed.Equals("P") ? true : false;
38             EnergySource = energySource.Equals("1") ? true : false;
39             EnergyType = (int)GetEnergyTypeFromString(energyType);
40             NumberTx = Convert.ToInt32(numberTx);
41             OnOff = onOff.Equals("1") ? true : false;
42             LastUpdate = DateTime.Now;
43         }
44
45         #endregion
46
47         #region "Public Properties and Enums"
48
49         public enum EnergyTypeEnum
50         {
51             Nothing = 0,
52             Battery = 1,
53             RF = 2,
54             Solar = 3,
55             USB = 4
56         };
57
58         /// <summary>
59         /// Gets or sets Bytes.
60         /// </summary>
61         public List<byte> Bytes { get; set; }
62
63         public string Id { get; set; }
64
65         public string Temperature { get; set; }
66
67         public string Voltage { get; set; }
68
69         public string RSSI { get; set; }
```

```
70
71          public bool ButtonPressed { get; set; }
72
73          public bool EnergySource { get; set; }
74
75          public int EnergyType { get; set; }
76
77          public bool OnOff { get; set; }
78
79          public int NumberTx { get; set; }
80
81          public DateTime LastUpdate { get; set; }
82
83          /// <summary>
84          /// Gets or sets CRC.
85          /// </summary>
86          public int CRC { get; set; }
87
88          /// <summary>
89          /// Gets or sets a value indicating whether Error.
90          /// </summary>
91          public bool Error { get; set; }
92
93          #endregion
94
95          #region "Public Methods"
96
97          public void Update(WSNNodeStatus newNodeStatus)
98          {
99              Id = newNodeStatus.Id;
100             Temperature = newNodeStatus.Temperature;
101             Voltage = newNodeStatus.Voltage;
102             RSSI = newNodeStatus.RSSI;
103             ButtonPressed = newNodeStatus.ButtonPressed;
104             EnergySource = newNodeStatus.EnergySource;
105             EnergyType = newNodeStatus.EnergyType;
106             NumberTx = newNodeStatus.NumberTx;
107             OnOff = newNodeStatus.OnOff;
108             LastUpdate = DateTime.Now;
109         }
110
111         public void Update(string id, string temperature, string voltage, string rssi, string ↙
    buttonPressed, int energyType, string energySource, string numberTx, string onOff)
112         {
113             Id = id;
114             Temperature = temperature;
115             Voltage = voltage;
116             RSSI = rssi;
117             EnergySource = energySource.Equals("1") ? true : false;
118             NumberTx = Convert.ToInt32(numberTx);
119             ButtonPressed = buttonPressed.Equals("P") ? true : false;
120             EnergyType = energyType;
121             OnOff = onOff.Equals("1") ? true : false;
122             LastUpdate = DateTime.Now;
123         }
124
125         public EnergyTypeEnum GetEnergyTypeFromString(string enerygType)
126         {
127             var tes = Convert.ToInt32(enerygType);
128
129             return (EnergyTypeEnum)tes;
130         }
131
132         public bool Equals(WSNNodeStatus other)
133         {
134             // Check whether the compared object is null.
135             if (Object.ReferenceEquals(other, null)) return false;
136
137             // Check whether the compared object references the same data.
138             if (Object.ReferenceEquals(this, other)) return true;
139
140             // Check whether the Ids are equal
```

```
141             return String.Compare(this.Id, other.Id) == 0 ? true : false;
142         }
143 
144         // If Equals returns true for a pair of objects,
145         // GetHashCode must return the same value for these objects.
146         public override int GetHashCode()
147         {
148             // Get the hash code for the Name field if it is not null.
149             int hashNodeID = this.Id == null ? 0 : this.Id.GetHashCode();
150 
151             // Calculate the hash code
152             return hashNodeID;
153         }
154 
155         /// <summary>
156         /// Prints node status.
157         /// </summary>
158         /// <returns>The node status.</returns>
159         public override string ToString()
160         {
161             return String.Concat(
162                 "Id ", Id, ' ',
163                 Temperature.Trim('C'), "ºC ",
164                 Voltage, "V ",
165                 RSSI, "% ",
166                 Enum.GetName(typeof(EnergyTypeEnum), EnergyType), ' ',
167                 ButtonPressed ? "P" : "NP", ' ',
168                 "ES: ", EnergySource ? "B" : "H",
169                 " Tx ", NumberTx,
170                 " TS ", LastUpdate, ' ',
171                 " TD ", Math.Round(DateTime.Now.Subtract(LastUpdate).TotalSeconds, 1));
172         }
173 
174         #endregion
175     }
176 }
```

D.8. Helpers

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Runtime.Serialization;
6  using System.Runtime.Serialization.Formatters.Binary;
7  using System.IO;
8
9  namespace WSN.ExtensionMethods
10 {
11     public static class WSNExtensionMethods
12     {
13         /// <summary>
14         /// Perform a deep Copy of the object.
15         /// </summary>
16         /// <typeparam name="T">The type of object being copied.</typeparam>
17         /// <param name="source">The object instance to copy.</param>
18         /// <returns>The copied object.</returns>
19         public static T Clone<T>(this T source)
20         {
21             if (!typeof(T).IsSerializable)
22             {
23                 throw new ArgumentException("The type must be serializable.", "source");
24             }
25
26             // Don't serialize a null object, simply return the default for that object
27             if (Object.ReferenceEquals(source, null))
28             {
29                 return default(T);
30             }
31
32             IFormatter formatter = new BinaryFormatter();
33             Stream stream = new MemoryStream();
34             using (stream)
35             {
36                 formatter.Serialize(stream, source);
37                 stream.Seek(0, SeekOrigin.Begin);
38                 return (T)formatter.Deserialize(stream);
39             }
40         }
41     }
42 }
43
```

Apéndice E

WSN Embedded Solution, Código Fuente

E.1. End Device

```
//******************************************************************************
// THIS PROGRAM IS PROVIDED "AS IS". TI MAKES NO WARRANTIES OR
// REPRESENTATIONS, EITHER EXPRESS, IMPLIED OR STATUTORY,
// INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS
// FOR A PARTICULAR PURPOSE, LACK OF VIRUSES, ACCURACY OR
// COMPLETENESS OF RESPONSES, RESULTS AND LACK OF NEGLIGENCE.
// TI DISCLAIMS ANY WARRANTY OF TITLE, QUIET ENJOYMENT, QUIET
// POSSESSION, AND NON-INFRINGEMENT OF ANY THIRD PARTY
// INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO THE PROGRAM OR
// YOUR USE OF THE PROGRAM.
//
// IN NO EVENT SHALL TI BE LIABLE FOR ANY SPECIAL, INCIDENTAL,
// CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY
// THEORY OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED
// OF THE POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT
// OF THIS AGREEMENT, THE PROGRAM, OR YOUR USE OF THE PROGRAM.
// EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED TO, COST OF
// REMOVAL OR REINSTALLATION, COMPUTER TIME, LABOR COSTS, LOSS
// OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF
// USE OR INTERRUPTION OF BUSINESS. IN NO EVENT WILL TI'S
// AGGREGATE LIABILITY UNDER THIS AGREEMENT OR ARISING OUT OF
// YOUR USE OF THE PROGRAM EXCEED FIVE HUNDRED DOLLARS
// (U.S.$500).
//
// Unless otherwise stated, the Program written and copyrighted
// by Texas Instruments is distributed as "freeware".  You may,
// only under TI's copyright in the Program, use and modify the
// Program without any charge or restriction.  You may
// distribute to third parties, provided that you transfer a
// copy of this license to the third party and the third party
// agrees to these terms by its first use of the Program. You
// must reproduce the copyright notice and any other legend of
// ownership on each copy or partial copy, of the Program.
//
// You acknowledge and agree that the Program contains
// copyrighted material, trade secrets and other TI proprietary
// information and is protected by copyright laws,
// international copyright treaties, and trade secret laws, as
// well as other intellectual property laws.  To protect TI's
// rights in the Program, you agree not to decompile, reverse
// engineer, disassemble or otherwise translate any object code
// versions of the Program to a human-readable form.  You agree
// that in no event will you alter, remove or destroy any
// copyright notice included in the Program.  TI reserves all
// rights not specifically granted under this license. Except
// as specifically provided herein, nothing in this agreement
// shall be construed as conferring by implication, estoppel,
// or otherwise, upon you, any license or other right under any
// TI patents, copyrights or trade secrets.
//
// You may not use the Program in non-TI devices.
//
//******************************************************************************
//   eZ430-RF2500 WSN Temperature Sensor End Device
//
//   Description: This is the End Device software for the eZ430-2500RF
//                WSN Temperature Sensing Demonstration
//
//
//   Z. Shivers, L. Vacirca, D. Iglesias
//   Version    1.05
//   Texas Instruments, Inc
//   July 2010
//   Known working builds:
//     IAR Embedded Workbench Kickstart (Version: 5.10.4)
//     Code Composer Studio (Version 4.1.2.00027)
//******************************************************************************
//Change Log:
//******************************************************************************
```

```
//Version:  1.06
//Comments: Added support for WSN application layer and connectivity with
//          MS Windows application software
//Version:  1.05
//Comments: Added support for various baud rates dependent on CPU frequencies
//Version:  1.04
//Comments: Added support for SimpliciTI 1.1.1
//          Moved radio wakeup in linkTo() to after ADC code to save power
//          Replaced delays with __delay_cylces() instrinsic
//          Replaced toggleLED with BSP functions
//          Added more comments
//Version:  1.03
//Comments: Added support for SimpliciTI 1.1.0
//          Added support for Code Composer Studio
//          Added security (Enabled with -DSMPL_SECURE in smpl_nwk_config.dat)
//          Added acknowledgement (Enabled with -DAPP_AUTO_ACK in smpl_nwk_config.dat)
//          Based the modifications on the AP_as_Data_Hub example code
//Version:  1.02
//Comments: Changed Port toggling to abstract method
//          Fixed comment typos
//Version:  1.01
//Comments: Added support for SimpliciTI 1.0.3
//          Added Flash storage/check of Random address
//          Moved LED toggle to HAL
//Version:  1.00
//Comments: Initial Release Version
//*****************************************************************************

#include "bsp.h"
#include "mrfi.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"
#include "vlo_rand.h"
#include "virtual_com_cmds.h"

/*-------------------------------------------------------------------------------
 * Global definitions
 *-------------------------------------------------------------------------------*/

/* How many times to try a TX and miss an acknowledge before doing a scan */
#define MISSES_IN_A_ROW  2

#define TES_Battery      0x31
#define TES_RF           0x32
#define TES_Solar        0x33
#define TES_USB          0x34

#define BUTTON_PRESSED    'P'
#define BUTTON_NOTPRESSED 'N'

/*-------------------------------------------------------------------------------
 * Prototypes
 *-------------------------------------------------------------------------------*/

static void linkTo(void);
void createRandomAddress(void);
__interrupt void ADC10_ISR(void);
__interrupt void Timer_A(void);
__interrupt void Button_ISR(void);

/*-------------------------------------------------------------------------------
 * Globals
 *-------------------------------------------------------------------------------*/

static uint8_t tes = TES_Battery;

static linkID_t sLinkID1 = 0;
```

```
/* Temperature offset set at production */
volatile int * tempOffset = (int *)0x10F4;
/* Initialize radio address location */
char * Flash_Addr = (char *)0x10F0;
/* Work loop semaphores */
static volatile uint8_t sSelfMeasureSem = 0;

/* Button presed state semaphores */
static volatile uint8_t sButtonPressed = 0;

/* Rx callback handler */
static uint8_t sRxCallback(linkID_t);

/*-------------------------------------------------------------------------------
 * Main
 *-------------------------------------------------------------------------------*/
void main (void)
{
  addr_t lAddr;

  /* Initialize board-specific hardware */
  BSP_Init();

  /* Check flash for previously stored address */
  if(Flash_Addr[0] == 0xFF && Flash_Addr[1] == 0xFF &&
     Flash_Addr[2] == 0xFF && Flash_Addr[3] == 0xFF)
  {
    createRandomAddress(); // Create and store a new random address
  }

  /* Read out address from flash */
  lAddr.addr[0] = Flash_Addr[0];
  lAddr.addr[1] = Flash_Addr[1];
  lAddr.addr[2] = Flash_Addr[2];
  lAddr.addr[3] = Flash_Addr[3];

  /* Tell network stack the device address */
  SMPL_Ioctl(IOCTL_OBJ_ADDR, IOCTL_ACT_SET, &lAddr);

  /* Initialize TimerA and oscillator */
  BCSCTL3 |= LFXT1S_2;                      // LFXT1 = VLO
  TACCTL0 = CCIE;                           // TACCR0 interrupt enabled

  P1IE |= 0x04;                             // P1.2 interrupt enabled
  P1IES |= 0x04;                            // P1.2 Hi/lo edge
  P1IFG &= ~0x04;                           // P1.2 IFG cleared

  /* This is conceptually very important!
  Timer module will generate an interrupt every second,
  in order to wake-up the MCU from sleep mode. This is the mechanism
  that enables us to create the LDCP (Low Duty Cycle Profile)
 */
  TACCR0 = 24000;                           // ~ 3 sec period
  TACTL = TASSEL_1 + MC_1;                  // ACLK, upmode

  /* Keep trying to join (a side effect of successful initialization) until
   * successful. Toggle LEDS to indicate that joining has not occurred.
   */
  while (SMPL_SUCCESS != SMPL_Init(sRxCallback))
  {
    BSP_TOGGLE_LED1();
    BSP_TOGGLE_LED2();
    /* Go to sleep (LPM3 with interrupts enabled)
     * Timer A0 interrupt will wake CPU up every second to retry initializing
     */
    __bis_SR_register(LPM3_bits+GIE);  // LPM3 with interrupts enabled
  }

  /* LEDs on solid to indicate successful join. */
```

```
  BSP_TURN_ON_LED1();
  BSP_TURN_ON_LED2();

  /* Unconditional link to AP which is listening due to successful join. */
  linkTo();

  while(1);
}

/*-------------------------------------------------------------------------------
 * Private Methods
 *-------------------------------------------------------------------------------*/

static void linkTo()
{
  uint8_t msg[MESSAGE_LENGTH];
#ifdef APP_AUTO_ACK
  uint8_t misses, done;
#endif

  /* Keep trying to link... */
  while (SMPL_SUCCESS != SMPL_Link(&sLinkID1))
  {
    BSP_TOGGLE_LED1();
    BSP_TOGGLE_LED2();
    /* Go to sleep (LPM3 with interrupts enabled)
     * Timer A0 interrupt will wake CPU up every second to retry linking
     */
    __bis_SR_register(LPM3_bits+GIE);
  }

  /* Turn off LEDs. */
  BSP_TURN_OFF_LED1();
  BSP_TURN_OFF_LED2();

  /* Put the radio to sleep */
  SMPL_Ioctl(IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SLEEP, 0);

  while (1)
  {
    /* Go to sleep, waiting for interrupt every second to acquire data */
    __bis_SR_register(LPM3_bits);

    /* Time to measure */
    if (sSelfMeasureSem) {
      volatile long temp;
      int degC, volt;
      int results[2];
#ifdef APP_AUTO_ACK
      uint8_t      noAck;
      smplStatus_t rc;
#endif

      /* Get temperature */
      ADC10CTL1 = INCH_10 + ADC10DIV_4;       // Temp Sensor ADC10CLK/5
      ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE + ADC10SR;
      /* Allow ref voltage to settle for at least 30us (30us * 8MHz = 240 cycles)
       * See SLAS504D for settling time spec
       */
      __delay_cycles(240);
      ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion start
      __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts enabled
      results[0] = ADC10MEM;                  // Retrieve result
      ADC10CTL0 &= ~ENC;

      /* Get voltage */
      ADC10CTL1 = INCH_11;                     // AVcc/2
      ADC10CTL0 = SREF_1 + ADC10SHT_2 + REFON + ADC10ON + ADC10IE + REF2_5V;
      __delay_cycles(240);
```

```
      ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion start
      __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrruts enable
      results[1] = ADC10MEM;                  // Retrieve results

      /* Stop and turn off ADC */
      ADC10CTL0 &= ~ENC;
      ADC10CTL0 &= ~(REFON + ADC10ON);

      /* oC = ((A10/1024)*1500mV)-986mV)*1/3.55mV = A10*423/1024 - 278
       * the temperature is transmitted as an integer where 32.1 = 321
       * hence 4230 instead of 423
       */
      temp = results[0];
      degC = ((temp - 673) * 4230) / 1024;
      if( (*tempOffset) != 0xFFFF )
      {
        degC += (*tempOffset);
      }

    /* message format,  UB = upper Byte, LB = lower Byte

-------------------------------------------------------------------------------------------------------
------------------
    |degC LB | degC UB |  volt LB | Mode  | # transmit LB |# transmit UB |  On/Off  |  Type of
Energy   |  Button State

-------------------------------------------------------------------------------------------------------
------------------
       0         1          2          3          4               5            6             7
8
    */

      temp = results[1];
      volt = (temp*25)/512;
      msg[0] = degC&0xFF;
      msg[1] = (degC>>8)&0xFF;
      msg[2] = volt;
      msg[3] = 110;
      msg[4] = 0;
      msg[5] = 0;
      msg[6] = 0;
      msg[7] = tes;

      /* Get radio ready...awakens in idle state */
      SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_AWAKE, 0);

#ifdef APP_AUTO_ACK
      /* Request that the AP sends an ACK back to confirm data transmission
       * Note: Enabling this section more than DOUBLES the current consumption
       *       due to the amount of time IN RX waiting for the AP to respond
       */
      done = 0;
      while (!done)
      {
        noAck = 0;

        /* Try sending message MISSES_IN_A_ROW times looking for ack */
        for (misses=0; misses < MISSES_IN_A_ROW; ++misses)
        {
          if (SMPL_SUCCESS == (rc=SMPL_SendOpt(sLinkID1, msg, sizeof(msg), SMPL_TXOPTION_ACKREQ)))
          {
            /* Message acked. We're done. Toggle LED 1 to indicate ack received. */
            BSP_TURN_ON_LED1();
            __delay_cycles(2000);
            BSP_TURN_OFF_LED1();
            break;
          }
          if (SMPL_NO_ACK == rc)
          {
```

```
            /* Count ack failures. Could also fail becuase of CCA and
             * we don't want to scan in this case.
             */
            noAck++;
          }
        }
        if (MISSES_IN_A_ROW == noAck)
        {
          /* Message not acked */
          BSP_TURN_ON_LED2();
          __delay_cycles(2000);
          BSP_TURN_OFF_LED2();
#ifdef FREQUENCY_AGILITY
          /* Assume we're on the wrong channel so look for channel by
           * using the Ping to initiate a scan when it gets no reply. With
           * a successful ping try sending the message again. Otherwise,
           * for any error we get we will wait until the next button
           * press to try again.
           */
          if (SMPL_SUCCESS != SMPL_Ping(sLinkID1))
          {
            done = 1;
          }
#else
          done = 1;
#endif  /* FREQUENCY_AGILITY */
        }
        else
        {
          /* Got the ack or we don't care. We're done. */
          done = 1;
        }
      }
#else

      if (sButtonPressed)
      {
        BSP_TURN_ON_LED2();
        msg[8] = BUTTON_PRESSED;
      }
      else
      {
        BSP_TURN_OFF_LED2();
        msg[8] = BUTTON_NOTPRESSED;
      }

      /* No AP acknowledgement, just send a single message to the AP */
      SMPL_SendOpt(sLinkID1, msg, sizeof(msg), SMPL_TXOPTION_NONE);

      /* Blink a LED to show the Tx */
      BSP_TURN_ON_LED1();
      __delay_cycles(6000);
      BSP_TURN_OFF_LED1();

#endif /* APP_AUTO_ACK */

      /* Put radio back to sleep */
      SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SLEEP, 0);

      /* Done with measurement, disable measure flag */
      sSelfMeasureSem = 0;
    }
  }
}

/* Handle received messages */
static uint8_t sRxCallback(linkID_t port)
{
  uint8_t msg[1], len;
```

```
  /* is the callback for the link ID we want to handle? */
  if (port == sLinkID1)
  {
    /* yes. go get the frame. we know this call will succeed. */
     if ((SMPL_SUCCESS == SMPL_Receive(sLinkID1, msg, &len)) && len)
     {
       /* Check the application sequence number to detect
        * late or missing frames...
        */

       // Get command id
       uint8_t command = (msg[0]>>6)&0x03;

       switch(command)
       {
         case 0x01:
         {
           uint8_t ledState = (msg[0]>>3)&0x01;

           if (ledState)
           {
             BSP_TURN_ON_LED2();
             BSP_TURN_ON_LED1();
           }
           else
           {
             BSP_TURN_OFF_LED2();
             BSP_TURN_OFF_LED1();
           }

           break;
         }
       }

       /* drop frame. we're done with it. */
       return 1;
     }
  }
  /* keep frame for later handling */
  return 0;
}

void createRandomAddress()
{
  unsigned int rand, rand2;
  do
  {
    rand = TI_getRandomIntegerFromVLO();    // first byte can not be 0x00 of 0xFF
  }
  while( (rand & 0xFF00)==0xFF00 || (rand & 0xFF00)==0x0000 );
  rand2 = TI_getRandomIntegerFromVLO();

  BCSCTL1 = CALBC1_1MHZ;                    // Set DCO to 1MHz
  DCOCTL = CALDCO_1MHZ;
  FCTL2 = FWKEY + FSSEL0 + FN1;             // MCLK/3 for Flash Timing Generator
  FCTL3 = FWKEY + LOCKA;                    // Clear LOCK & LOCKA bits
  FCTL1 = FWKEY + WRT;                      // Set WRT bit for write operation

  Flash_Addr[0]=(rand>>8) & 0xFF;
  Flash_Addr[1]=rand & 0xFF;
  Flash_Addr[2]=(rand2>>8) & 0xFF;
  Flash_Addr[3]=rand2 & 0xFF;

  FCTL1 = FWKEY;                            // Clear WRT bit
  FCTL3 = FWKEY + LOCKA + LOCK;             // Set LOCK & LOCKA bit
}

/*-------------------------------------------------------------------------
```

```
 * Interrupt Service Routines
 *-------------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------------
 * ADC10 interrupt service routine
 *-------------------------------------------------------------------------------*/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
  __bic_SR_register_on_exit(CPUOFF);        // Clear CPUOFF bit from 0(SR)
}

/*-------------------------------------------------------------------------------
 * Timer A0 interrupt service routine
 *-------------------------------------------------------------------------------*/
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  sSelfMeasureSem = 1;
  __bic_SR_register_on_exit(LPM3_bits);        // Clear LPM3 bit from 0(SR)
}

/*-------------------------------------------------------------------------------
 * Button-pressed interrupt service routine
 *-------------------------------------------------------------------------------*/
#pragma vector=PORT1_VECTOR
__interrupt void Button_ISR(void)
{
  sButtonPressed = !sButtonPressed;

  // Clear the interrupt flag
  P1IFG &= ~0x04;
}
```

E.2. Access Point

```
//*****************************************************************************
// THIS PROGRAM IS PROVIDED "AS IS". TI MAKES NO WARRANTIES OR
// REPRESENTATIONS, EITHER EXPRESS, IMPLIED OR STATUTORY,
// INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS
// FOR A PARTICULAR PURPOSE, LACK OF VIRUSES, ACCURACY OR
// COMPLETENESS OF RESPONSES, RESULTS AND LACK OF NEGLIGENCE.
// TI DISCLAIMS ANY WARRANTY OF TITLE, QUIET ENJOYMENT, QUIET
// POSSESSION, AND NON-INFRINGEMENT OF ANY THIRD PARTY
// INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO THE PROGRAM OR
// YOUR USE OF THE PROGRAM.
//
// IN NO EVENT SHALL TI BE LIABLE FOR ANY SPECIAL, INCIDENTAL,
// CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY
// THEORY OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED
// OF THE POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT
// OF THIS AGREEMENT, THE PROGRAM, OR YOUR USE OF THE PROGRAM.
// EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED TO, COST OF
// REMOVAL OR REINSTALLATION, COMPUTER TIME, LABOR COSTS, LOSS
// OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF
// USE OR INTERRUPTION OF BUSINESS. IN NO EVENT WILL TI'S
// AGGREGATE LIABILITY UNDER THIS AGREEMENT OR ARISING OUT OF
// YOUR USE OF THE PROGRAM EXCEED FIVE HUNDRED DOLLARS
// (U.S.$500).
//
// Unless otherwise stated, the Program written and copyrighted
// by Texas Instruments is distributed as "freeware".  You may,
// only under TI's copyright in the Program, use and modify the
// Program without any charge or restriction.  You may
// distribute to third parties, provided that you transfer a
// copy of this license to the third party and the third party
// agrees to these terms by its first use of the Program. You
// must reproduce the copyright notice and any other legend of
// ownership on each copy or partial copy, of the Program.
//
// You acknowledge and agree that the Program contains
// copyrighted material, trade secrets and other TI proprietary
// information and is protected by copyright laws,
// international copyright treaties, and trade secret laws, as
// well as other intellectual property laws.  To protect TI's
// rights in the Program, you agree not to decompile, reverse
// engineer, disassemble or otherwise translate any object code
// versions of the Program to a human-readable form.  You agree
// that in no event will you alter, remove or destroy any
// copyright notice included in the Program.  TI reserves all
// rights not specifically granted under this license. Except
// as specifically provided herein, nothing in this agreement
// shall be construed as conferring by implication, estoppel,
// or otherwise, upon you, any license or other right under any
// TI patents, copyrights or trade secrets.
//
// You may not use the Program in non-TI devices.
//
//*****************************************************************************
//   eZ430-RF2500 Temperature Sensor Access Point
//
//   Description: This is the End Device software for the eZ430-2500RF
//                WSN Temperature Sensing Demonstration
//
//
//   Z. Shivers, L. Vacirca, D. Iglesias
//   Version    1.05
//   Texas Instruments, Inc
//   July 2010
//   Known working builds:
//     IAR Embedded Workbench Kickstart (Version: 5.10.4)
//     Code Composer Studio (Version 4.1.2.00027)
//*****************************************************************************
//Change Log:
//*****************************************************************************
```

```
//Version:  1.06
//Comments: Added support for WSN application layer and connectivity with
//          MS Windows application software
//Version:  1.05
//Comments: Added support for various baud rates dependent on CPU frequencies
//Version:  1.04
//Comments: Added support for SimpliciTI 1.1.1
//          Moved radio wakeup in linkTo() to after ADC code to save power
//          Replaced delays with __delay_cylces() instrinsic
//          Replaced toggleLED with BSP functions
//          Added more comments
//Version:  1.03
//Comments: Added support for SimpliciTI 1.1.0
//          Added support for Code Composer Studio
//          Added security (Enabled with -DSMPL_SECURE in smpl_nwk_config.dat)
//          Added acknowledgement (Enabled with -DAPP_AUTO_ACK in smpl_nwk_config.dat)
//          Based the modifications on the AP_as_Data_Hub example code
//Version:  1.02
//Comments: Changed Port toggling to abstract method
//          Fixed comment typos
//Version:  1.01
//Comments: Added support for SimpliciTI 1.0.3
//          Added Flash storage/check of Random address
//          Moved LED toggle to HAL
//Version:  1.00
//Comments: Initial Release Version
//*****************************************************************************
#include <string.h>
#include "bsp.h"
#include "mrfi.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "nwk_frame.h"
#include "nwk.h"
#include "virtual_com_cmds.h"

/****************** COMMENTS ON ASYNC LISTEN APPLICATION ***********************
Summary:
  This AP build includes implementation of an unknown number of end device peers
  in addition to AP functionality. In this scenario all End Devices establish a
  link to the AP and only to the AP. The AP acts as a data hub. All End Device
  peers are on the AP and not on other distinct ED platforms.

  There is still a limit to the number of peers supported on the AP that is
  defined by the macro NUM_CONNECTIONS. The AP will support NUM_CONNECTIONS or
  fewer peers but the exact number does not need to be known at build time.

  In this special but common scenario SimpliciTI restricts each End Device
  object to a single connection to the AP. If multiple logical connections are
  required these must be accommodated by supporting contexts in the application
  payload itself.

Solution overview:
  When a new peer connection is required the AP main loop must be notified. In
  essence the main loop polls a semaphore to know whether to begin listening for
  a peer Link request from a new End Device. There are two solutions: automatic
  notification and external notification. The only difference between the
  automatic notification solution and the external notification solution is how
  the listen semaphore is set. In the external notification solution the
  sempahore is set by the user when the AP is stimulated for example by a button
  press or a commend over a serial link. In the automatic scheme the
  notification is accomplished as a side effect of a new End Device joining.

  The Rx callback must be implemented. When the callback is invoked with a
  non-zero Link ID the handler could set a semaphore that alerts the main work
  loop that a SMPL_Receive() can be executed successfully on that Link ID.
```

```
  If the callback conveys an argument (LinkID) of 0 then a new device has joined
  the network. A SMPL_LinkListen() should be executed.

  Whether the joining device supports ED objects is indirectly inferred on the
  joining device from the setting of the NUM_CONNECTIONS macro. The value of
  this macro should be non-zero only if ED objects exist on the device. This
  macro is always non-zero for ED-only devices. But Range Extenders may or may
  not support ED objects. The macro should be be set to 0 for REs that do not
  also support ED objects. This prevents the Access Point from reserving
  resources for a joinng device that does not support any End Device Objects and
  it prevents the AP from executing a SMPL_LinkListen(). The Access Point will
  not ever see a Link frame if the joining device does not support any
  connections.

  Each joining device must execute a SMPL_Link() after receiving the join reply
  from the Access Point. The Access Point will be listening.

****************** END COMMENTS ON ASYNC LISTEN APPLICATION *****************/

/******  THIS SOURCE FILE REPRESENTS THE AUTOMATIC NOTIFICATION SOLUTION ******/

/*-------------------------------------------------------------------------------
 * Prototypes
 *-------------------------------------------------------------------------------*/

/* Frequency Agility helper functions */
static void    checkChangeChannel(void);
static void    changeChannel(void);

__interrupt void ADC10_ISR(void);
__interrupt void Timer_A (void);

/*-------------------------------------------------------------------------------
 * Globals
 *-------------------------------------------------------------------------------*/

#define TES_Battery      0x31
#define TES_RF           0x32
#define TES_Solar        0x33
#define TES_USB          0x34

#define BUTTON_PRESSED    'P'
#define BUTTON_NOTPRESSED 'N'

/* Type of Energy Source */
static uint8_t tes = TES_USB;

/* reserve space for the maximum possible peer Link IDs */
static linkID_t sLID[NUM_CONNECTIONS] = {0};
static uint8_t  sNumCurrentPeers = 0;

/* callback handler */
static uint8_t sCB(linkID_t);

/* received message handler */
static void processMessage(linkID_t, uint8_t *, uint8_t);

/* work loop semaphores */
static volatile uint8_t sPeerFrameSem = 0;
static volatile uint8_t sJoinSem = 0;
static volatile uint8_t sSelfMeasureSem = 0;

extern volatile uint8_t sReceivedCommand;
extern volatile uint8_t sReceivedMessage[1];

/* blink LEDs when channel changes... */
static volatile uint8_t sBlinky = 0;

/* data for terminal output */
```

```
volatile int * tempOffset = (int *)0x10F4;

/*-------------------------------------------------------------------------------
 * Frequency Agility support (interference detection)
 *-------------------------------------------------------------------------------*/
#ifdef FREQUENCY_AGILITY

#define INTERFERNCE_THRESHOLD_DBM (-70)
#define SSIZE    25
#define IN_A_ROW  3
static int8_t  sSample[SSIZE];
static uint8_t sChannel = 0;

#endif  /* FREQUENCY_AGILITY */

/*-------------------------------------------------------------------------------
 * Main
 *-------------------------------------------------------------------------------*/
void main (void)
{
  bspIState_t intState;

#ifdef FREQUENCY_AGILITY
  memset(sSample, 0x0, sizeof(sSample));
#endif

  /* Initialize board */
  BSP_Init();

  /* Initialize TimerA and oscillator */
  BCSCTL3 |= LFXT1S_2;                      // LFXT1 = VLO
  TACCTL0 = CCIE;                           // TACCR0 interrupt enabled
  TACCR0 = 10000;                           // ~1 second
  TACTL = TASSEL_1 + MC_1;                  // ACLK, upmode

  sReceivedCommand = 0;

  /* Initialize serial port */
  COM_Init();

  SMPL_Init(sCB);

  /* green and red LEDs on solid to indicate waiting for a Join. */
  BSP_TURN_ON_LED1();
  BSP_TURN_ON_LED2();

  /* main work loop */
  while (1)
  {
    /* Wait for the Join semaphore to be set by the receipt of a Join frame from
     * a device that supports an End Device.
     *
     * An external method could be used as well. A button press could be connected
     * to an ISR and the ISR could set a semaphore that is checked by a function
     * call here, or a command shell running in support of a serial connection
     * could set a semaphore that is checked by a function call.
     */
    if (sJoinSem && (sNumCurrentPeers < NUM_CONNECTIONS))
    {
      /* listen for a new connection */
      while (1)
      {
        if (SMPL_SUCCESS == SMPL_LinkListen(&sLID[sNumCurrentPeers]))
        {
          break;
        }
        /* Implement fail-to-link policy here. otherwise, listen again. */
      }
```

```
  sNumCurrentPeers++;

  BSP_ENTER_CRITICAL_SECTION(intState);
  sJoinSem--;
  BSP_EXIT_CRITICAL_SECTION(intState);
}


// if it is time to measure our own temperature...
if(sSelfMeasureSem)
{
  char msg [MESSAGE_LENGTH];
  char addr[] = {"HUB0"};
  char rssi[] = {"000"};
  int degC, volt;
  volatile long temp;
  int results[2];

  /* Get temperature */
  ADC10CTL1 = INCH_10 + ADC10DIV_4;       // Temp Sensor ADC10CLK/5
  ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE + ADC10SR;
  /* Allow ref voltage to settle for at least 30us (30us * 8MHz = 240 cycles)
   * See SLAS504D for settling time spec
   */
  __delay_cycles(240);
  ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion start
  __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts enabled
  results[0] = ADC10MEM;                  // Retrieve result
  ADC10CTL0 &= ~ENC;

  /* Get voltage */
  ADC10CTL1 = INCH_11;                     // AVcc/2
  ADC10CTL0 = SREF_1 + ADC10SHT_2 + REFON + ADC10ON + ADC10IE + REF2_5V;
  __delay_cycles(240);
  ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion start
  __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts enabled
  results[1] = ADC10MEM;                  // Retrieve result

  /* Stop and turn off ADC */
  ADC10CTL0 &= ~ENC;
  ADC10CTL0 &= ~(REFON + ADC10ON);

  /* oC = ((A10/1024)*1500mV)-986mV)*1/3.55mV = A10*423/1024 - 278
   * the temperature is transmitted as an integer where 32.1 = 321
   * hence 4230 instead of 423
   */
  temp = results[0];
  degC = ((temp - 673) * 4230) / 1024;
  if( (*tempOffset) != 0xFFFF )
  {
    degC += (*tempOffset);
  }

  temp = results[1];
  volt = (temp*25)/512;

  /* Package up the data */
  msg[0] = degC&0xFF;
  msg[1] = (degC>>8)&0xFF;
  msg[2] = volt;
  msg[4] = 0;
  msg[5] = 0;
  msg[6] = 0;
  msg[7] = tes;
  msg[8] = BUTTON_NOTPRESSED;

  /* Send it over serial port */
  transmitDataString(addr, rssi, msg );
```

```
  BSP_TOGGLE_LED1();

  /* Done with measurement, disable measure flag */
  sSelfMeasureSem = 0;
}

/* Have we received a frame on one of the ED connections?
 * No critical section -- it doesn't really matter much if we miss a poll
 */
if (sPeerFrameSem)
{
  uint8_t     msg[MAX_APP_PAYLOAD], len, i;

  /* process all frames waiting */
  for (i=0; i<sNumCurrentPeers; ++i)
  {
    if (SMPL_SUCCESS == SMPL_Receive(sLID[i], msg, &len))
    {
      ioctlRadioSiginfo_t sigInfo;

      processMessage(sLID[i], msg, len);

      sigInfo.lid = sLID[i];

      SMPL_Ioctl(IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SIGINFO, (void *)&sigInfo);

      transmitData( i, (signed char)sigInfo.sigInfo.rssi, (char*)msg );
      BSP_TOGGLE_LED2();

      BSP_ENTER_CRITICAL_SECTION(intState);
      sPeerFrameSem--;
      BSP_EXIT_CRITICAL_SECTION(intState);
    }
  }
}

if (sReceivedCommand)
{
  uint8_t nodeID = sReceivedMessage[0] & 0x03;
  uint8_t linkID = sLID[nodeID];
  uint8_t message[1];

  message[0] = sReceivedMessage[0];

  SMPL_Send(linkID, message, sizeof(message));

  /* Done with command processing, disable flag */
  sReceivedMessage[0] = '\0';
  sReceivedCommand = 0;
}

if (BSP_BUTTON1())
{
  __delay_cycles(2000000);  /* debounce (0.25 seconds) */
  changeChannel();
}
else
{
  checkChangeChannel();
}
BSP_ENTER_CRITICAL_SECTION(intState);
if (sBlinky)
{
  if (++sBlinky >= 0xF)
  {
    sBlinky = 1;
    BSP_TOGGLE_LED1();
    BSP_TOGGLE_LED2();
  }
```

```
    }
    BSP_EXIT_CRITICAL_SECTION(intState);
  }

}

/* Runs in ISR context. Reading the frame should be done in the */
/* application thread not in the ISR thread. */
static uint8_t sCB(linkID_t lid)
{
  if (lid)
  {
    sPeerFrameSem++;
    sBlinky = 0;
  }
  else
  {
    sJoinSem++;
  }

  /* leave frame to be read by application. */
  return 0;
}

static void processMessage(linkID_t lid, uint8_t *msg, uint8_t len)
{
  /* do something useful */
  if (len)
  {
    BSP_TOGGLE_LED1();
  }
  return;
}

static void changeChannel(void)
{
#ifdef FREQUENCY_AGILITY
  freqEntry_t freq;

  if (++sChannel >= NWK_FREQ_TBL_SIZE)
  {
    sChannel = 0;
  }
  freq.logicalChan = sChannel;
  SMPL_Ioctl(IOCTL_OBJ_FREQ, IOCTL_ACT_SET, &freq);
  BSP_TURN_OFF_LED1();
  BSP_TURN_OFF_LED2();
  sBlinky = 1;
#endif
  return;
}

/* implement auto-channel-change policy here... */
static void checkChangeChannel(void)
{
#ifdef FREQUENCY_AGILITY
  int8_t dbm, inARow = 0;

  uint8_t i;

  memset(sSample, 0x0, SSIZE);
  for (i=0; i<SSIZE; ++i)
  {
    /* quit if we need to service an app frame */
    if (sPeerFrameSem || sJoinSem)
    {
      return;
    }
    NWK_DELAY(1);
```

```
    SMPL_Ioctl(IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RSSI, (void *)&dbm);
    sSample[i] = dbm;

    if (dbm > INTERFERNCE_THRESHOLD_DBM)
    {
      if (++inARow == IN_A_ROW)
      {
        changeChannel();
        break;
      }
    }
    else
    {
      inARow = 0;
    }
  }
#endif
  return;
}

/*-------------------------------------------------------------------------------
* ADC10 interrupt service routine
-------------------------------------------------------------------------------*/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
  __bic_SR_register_on_exit(CPUOFF);        // Clear CPUOFF bit from 0(SR)
}

/*-------------------------------------------------------------------------------
* Timer A0 interrupt service routine
-------------------------------------------------------------------------------*/
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  sSelfMeasureSem = 1;
}
```