

UNIVERSIDAD POLITÉCNICA DE CARTAGENA



PROYECTO FIN DE CARRERA

Implementación de Aplicaciones Sensoriales en el dispositivo TelosB sobre IEEE 802.15.4



AUTOR: Pedro José López Sánchez
DIRECTORES: Antonio Javier García Sánchez
Felipe García Sánchez



Autor Pedro J. López Sánchez
E-mail pedroilopezsanchez@gmail.com

Director Antonio Javier García Sánchez
Felipe García Sanchez

E-mail antoniojavier.garcia@upct.es
felipe.garcia@upct.es

Título del Proyecto: Implementación de Aplicaciones Sensoriales en el dispositivo TelosB sobre IEEE 802.15.4

Resumen

Las redes inalámbricas de tipo personal (WPAN-Wireless Personal Area Networks) tienen como objetivo comunicar diferentes dispositivos inalámbricos en los más diversos entornos. Así podremos encontrarlas en nuestros hogares, en entornos destinados a la rama sanitaria o en centros de investigación y fábricas.

Estas redes personales no suelen estar comunicadas con el exterior, ya que se usarán principalmente para monitorización de procesos, seguridad, muestreo de señales, recepción de video o audio, y muchas otras tareas de control que no suele ser necesario interconectar con el exterior.

La cobertura de dichos dispositivos constituidos en redes personales (PAN) no es muy grande, lo que las hace diferentes de las redes inalámbricas locales (LAN), con mayor cobertura. Pero estos dispositivos inalámbricos hacen un uso muy eficiente de las comunicaciones. Por lo que tienen un bajo consumo, y también un bajo coste.

En este Proyecto Fin de Carrera se mostrará al lector los fundamentos de las redes basadas en dispositivos sensores de baja potencia usando el sistema operativo TinyOS, el muestreo de sensores de los dispositivos y posterior comunicación con otros nodos de la red. Se detallará la tecnología, se comentarán sus características principales y la problemática que se plantea en este Proyecto Fin de Carrera. Se describirá el estándar 802.15.4, así como su funcionamiento, y se planteará una implementación 802.15.4 como solución a los inconvenientes de TinyOS.

Agradecimientos

Me gustaría agradecer a mi director de proyecto Antonio, primero por su paciencia conmigo y también por haberme animado a estudiar y trabajar en un proyecto con el cual tuviera una formación bastante buena sobre nuevos protocolos de comunicaciones; una iniciación en una posible rama de trabajo, que son las redes de sensores inalámbricos de baja potencia. Siempre que he acudido a él me ha atendido con gran amabilidad y me ha dado todo lo necesario para el trabajo de campo y la documentación que pudiera necesitar. Gracias Antonio.

Quiero agradecer también a mi compañero de carrera Jesús Cerezuela por haber sido mi “escudero” en las horas y horas de trabajo en la Biblioteca de Antiguones y sobre todo esas charlas en los “descansos” para seguir trabajando con más fuerza.

Gracias también a mis amigos Jesús, Guti, Jose, Rubén (¡a ver cuando acabas tu proyecto Rubén!), Lorenzo y Jorge, porque siempre han estado ahí para transmitirme el apoyo para continuar en la lucha.

A mi familia, mi madre, por su paciencia conmigo, y a toda la gente que ha confiado en mi sabiendo que terminaría la carrera.

Y finalmente Maritza, por enseñarme a manejar Word como Dios manda, por su paciencia eterna y esperar de mí lo mejor, Gracias.

Índice General

Agradecimientos	5
Capítulo 1 – Introducción	8
Capítulo 2 – Estado del Arte	9
1. Estandar 802.15.4	9
1.1 Introducción	9
1.2 Capas de Red	10
1.3 Capa de enlace de datos (Data Link Layer, DLL).	11
1.3.1 Formato general de tramas MAC.	12
1.3.2 La estructura de las super-tramas.	13
1.4 Capa física.	14
1.5 Canales para las comunicaciones.	15
1.6 Estructura de paquetes de información	16
1.7 Modulación	17
1.8 Sensitividad y Rango	18
1.9 Interferencia de y para otros dispositivos	18
2. TinyOS	19
2.1 Introducción	19
2.2 Implementación	19
2.2.1 NesC	20
2.3 Tutoriales de TinyOS	22
2.4 Open-Zigbee	23
2.4.1 Funcionalidades de Open-Zigbee	24
2.4.2 Estructura de la implementación	24
2.4.3 Implementación de la Capa Física y MAC	27
2.4.3.1 Implementación de la capa Física	27
2.4.3.2 Implementación de la capa MAC	28
2.4.4 Open-Zigbee 2.0	30
3. Tecnología	30
3.1 TelosB	30
3.2 Sensirion SHT-11	31
3.2.1 Sensirion SHT-11 dentro de TinyOS	32
3.2.2 Abstracción del Hardware. HPL, HAL y HIL.	33

Capítulo 3 – Implementación Red de Sensores	36
1. Desarrollo sobre TinyOS 2.0.2	36
1.1 BaseStation	37
1.2 Oscilloscope	46
1.3 SerialForwarder	55
1.4 Herramienta Java de Oscilloscope	56
2. Problemática de TinyOS 2.0.2	57
3. Desarrollo con Open-Zigbee 2.0 Beta	59
3.1 Transmisión de mensajes por medio de Active Message.	59
3.2 Interfaz de envío de datos en Open-Zigbee: MCPS_DATA	60
3.3 Oscilloscope Open-Zigbee	61
3.4 BaseStation Open-Zigbee	70
3.5 Conclusiones	76
Capítulo 4 -Resultados del proyecto	85
Conclusiones y trabajos futuros	92
Bibliografía	93
Apéndice A: Instalación de Cygwin y TinyOS	96

Índice de Figuras

- Fig.1 - Esquema de distintos tipos de redes inalámbricas
- Fig.2 - Ejemplo de topologías en estrella y peer-to-peer
- Fig.3 - Relación de capas en 802.15.4
- Fig.4 - Formato de trama MAC
- Fig.5 - Estructura de una supertrama 802.15.4
- Fig.6 - Estructura de los canales en 802.15.4
- Fig.7 - Frecuencia de los canales en 802.15.4
- Fig.8 - Paquete PHY
- Fig. 9 - Ejemplo de aplicación con varias componentes
- Fig.10 - Arquitectura de la pila del protocolo
- Fig.11 - Modelo de referencia de la capa física
- Fig.12 - Modelo de referencia de la capa MAC
- Fig.13 - Diagrama del dispositivo TelosB
- Fig.14 - Sensor Temperatura/Humedad SHT-11
- Fig.15 - Fórmula para el cálculo de la humedad relativa
- Fig.16 - Gráfico de conversión de Humedad Relativa
- Fig.17 - Arquitectura de la Abstracción del Hardware propuesta
- Fig.18 - Esquema de desarrollo de la aplicación
- Fig.19 – Consola de Cygwin compilación BaseStation
- Fig. 20 – Consola de Cygwin compilación Oscilloscope
- Fig.21 - GUI (Graphic User Interface) de Serial Forwarder
- Fig.22 - Vista principal de la GUI de la aplicación Java de Osciloscopio
- Fig.23 - Dependencias de componentes para el uso del chip de radio CC2420 en TinyOS

Índice de Tablas

- Tabla 1 – Propiedades del IEEE 802.15.4
- Tabla 2 – Parámetros de modulación
- Tabla 3 – Tabla de conversión de temperatura SHT-11

1 Introducción.

Las redes de sensores inalámbricas (WSN, Wireless Sensor Network) están formadas por un conjunto de dispositivos que permiten comunicarse sin cables, interconectados entre sí a través de una red inalámbrica y a su vez conectados a un sistema central, en el que se recopilará la información recogida por cada uno de los sensores, o bien redes tipo cluster peer-to-peer, en las cuales no hay un dispositivo que actúe de coordinador central.

Gracias a la investigación en nuevas tecnologías se han desarrollado nuevos dispositivos de pequeño tamaño que son capaces de formar redes inalámbricas, mediante el uso de sensores que toman variables de todo tipo (luz, humedad, temperatura, etc.) transmitirse información, video comprimido, audio, etc. Dichos dispositivos que poseen un microcontrolador, interfaz radio transmisor/receptor, memoria y sensores varios, constituyen los nodos, que se interconectan a su vez para formar redes. A cada nodo generalmente se le conoce comúnmente como *mote*.

El trabajo en este proyecto está basado en sensores de tecnología Zigbee-IEEE 802.15.4, los cuales tienen entre varias bondades ser capaces de crear redes de bajo coste, la flexibilidad de la red y el bajo consumo de energía.

Como posibles contras que posee la tecnología Zigbee está la limitación de tamaño de la trama de envío de datos. Tenemos tasas bajas de envío de datos precisamente en busca del ahorro de energía. También el alcance será limitado para evitar el mayor consumo de energía.

Por todo ello esta tecnología será bien aceptada en diversos campos como la domótica, automoción, monitorización de todo tipo (agrícola, industrial o sanitaria) y ha entrado a competir con otros tipos de tecnologías que monopolizaban las redes inalámbricas (WiFi, Bluetooth, etc). Por ello, en este PFC se da a conocer y formar al lector en los nuevos tipos de protocolos desarrollados como alternativa a los ya existentes.

Muchas de las aplicaciones que se desarrollan sobre esta tecnología están basadas en la plataforma TinyOS [1]. Tinyos ofrece librerías y aplicaciones de gran utilidad a los usuarios. Pero Tinyos tiene una desventaja. Aunque usa tamaños de trama y formatos de trama típicos de la tecnología IEEE 802.15.4, su protocolo de acceso al medio está basado en Wifi.

El objetivo de este PFC es ofrecer al usuario un nuevo marco de desarrollo de aplicaciones típicas de las WSN. Para ello se hará uso de la pila de protocolos completa IEEE 802.15.4, que permite optimizar al máximo los recursos que nos ofrece esta tecnología.

Se ha seleccionado el dispositivo TelosB [2] fabricado por la empresa Crossbow [3], que posee varios sensores (temperatura, humedad, etc.) de monitorización y módulos de comunicaciones IEEE 802.15.4.

2 Estado del Arte.

1. Estandar 802.15.4

1.1 Introducción.

Las principales características en este estándar son los bajos costos, su flexibilidad de red, mínimo consumo de energía; este estándar se utiliza para aplicaciones que requieren una baja tasa de transmisión de datos.

El motivo del uso de tecnología inalámbrica es la reducción en los gastos de instalación, ya que no requiere cambiar el cableado. Las redes inalámbricas suponen pues el intercambio de información con un mínimo de esfuerzo de instalación. Esta tendencia es impulsada por la gran capacidad de integrar componentes inalámbricos de una forma más barata y el éxito que tienen otros sistemas de comunicación inalámbrica como la telefonía móvil.

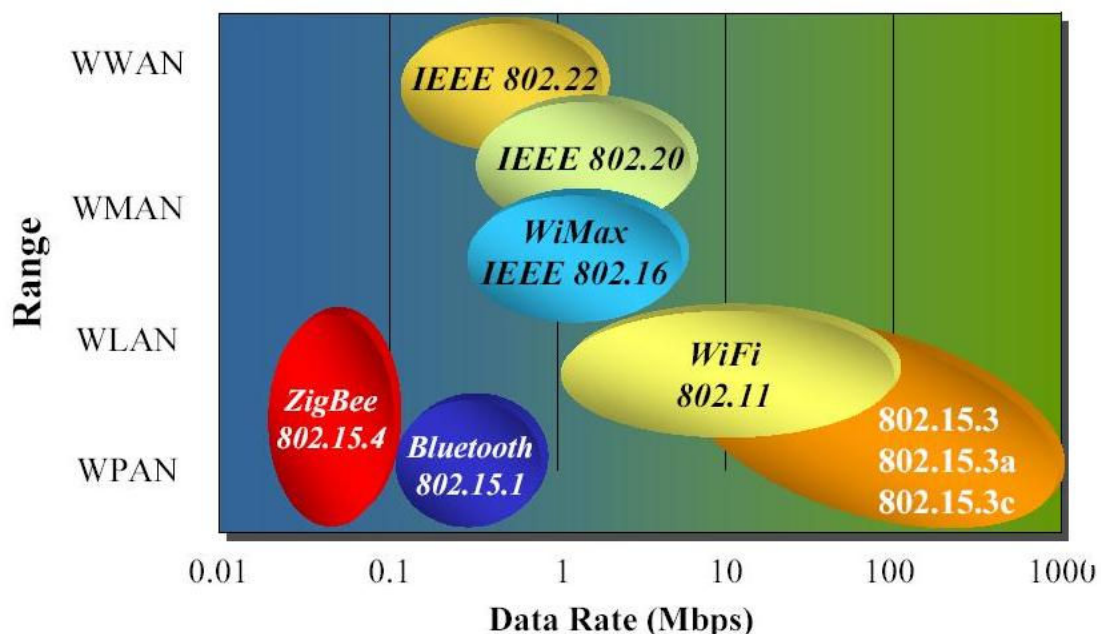


Figura 1- Esquema de distintos tipos de redes inalámbricas

En el año 2000 dos grupos especialistas en estándares (ZigBee y el grupo de trabajo IEEE 802) se unieron para dar a conocer la necesidad de un nuevo estándar para redes

inalámbricas de bajo consumo y de bajo coste para aplicaciones domóticas e industriales. Dando como resultado un nuevo estándar para áreas personales (LR-WPAN, Low Range Wíreles Personal Area Network) que ahora se conoce como el 802.15.4, más conocido como ZigBee.

Algunas características del 802.15.4 se resumen en la Tabla 1:

Propiedad	Valor
Rango de transmisión de datos	868 MHz: 20kb/s; 915 MHz: 40kb/s; 2.4 GHz: 250 kb/s.
Alcance	10 – 20 m.
Latencia	Por debajo de 15 ms.
Canales	868/915 MHz: 11 canales. 2.4 GHz: 16 canales.
Bandas de frecuencia	Dos PHY: 868/915 MHz y 2.4 GHz.
Direccionamiento	Corto de 8 bits o 64 bits IEEE
Canal de Acceso	CSMA-CA y CSMA-CA ranurado
Temperatura	El rango de temperatura industrial: -40° a +85° C

Tabla 1 - Propiedades del IEEE 802.15.4

1.2 Capas de Red.

En las redes cableadas tradicionales, la capa de red es la responsable de la topología y el mantenimiento de la misma. Asimismo también suele encargarse de tareas de direccionamiento y de seguridad.

Lo mismo existe para las redes inalámbricas, pero supone un reto mayor por la optimización para el ahorro de energía en las mismas.

Las redes que se construyan dentro de esta capa del estándar IEEE 802.15.4 se espera que se auto organicen y se auto mantengan en funcionamiento con lo que se pretende reducir los costos totales para el consumidor.

El estándar IEEE 802.15.4 soporta distintas topologías para formar una red, entre ellas la topología tipo estrella y la topología peer-to-peer.

La topología a escoger es una elección de diseño y dependerá de la aplicación a la que se desee orientar; por ejemplo, una topología en estrella para distintos periféricos conectados a un nodo principal conectado al PC, o una topología peer-to-peer con distintos nodos interconectados para establecer un perímetro de seguridad de amplia cobertura. La Figura 2 muestra ambos tipos de topologías.

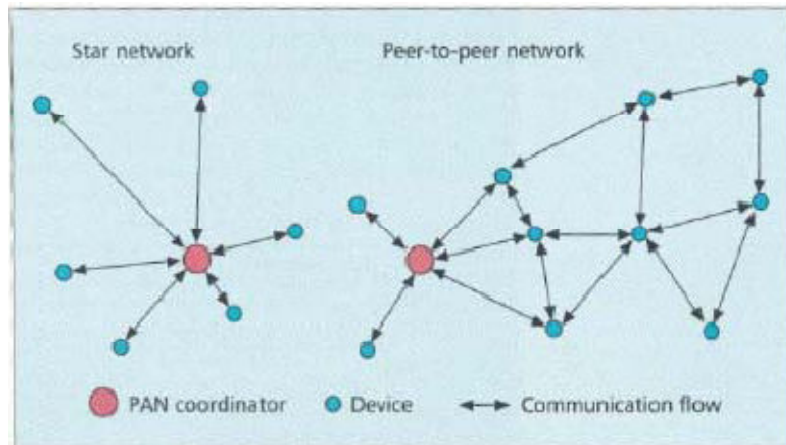


Figura 2 - Ejemplo de topologías en estrella y peer-to-peer

1.3 Capa de enlace de datos (Data Link Layer, DLL).

El proyecto IEEE 802 divide al DLL en dos sub capas, la sub capa de enlace de acceso a medios (Medium Access Control, MAC) y la de control de enlaces lógicos (Logical link control, LLC). El LLC es común a todos estándares 802, mientras que la sub capa MAC depende del hardware y varía respecto a la implementación física de esta capa. La Figura 3 ilustra la forma en que el estándar IEEE 802.15.4 se basa en la organización internacional para la estandarización (ISO) del modelo de referencia para la interconexión de sistemas abiertos (OSI).

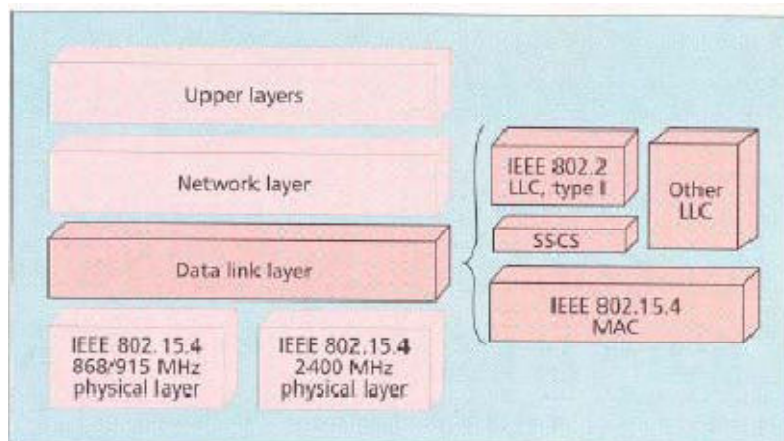


Figura 3 - Relacion de capas en 802.15.4

Las características del MAC IEEE 802.15.4 son: la asociación y la disociación, reconocimientos de entrega de trama, mecanismos de acceso al canal, validación de trama, garantía del manejo de las ranuras de tiempo, y manejo de guías. Las sub capas MAC proporcionan dos tipos de servicios hacia capas superiores que se acceden a través de dos puntos de acceso a servicios (SAPs). Los servicios de datos MAC se acceden por medio de la parte común de la sub capa (MCPS-SAP), y el manejo de servicios MAC se accede por medio de la capa MAC de manejo de identidades (MLME-

SAP). Esos dos servicios proporcionan una interfase entre las sub capas de convergencia de servicios específicos (SSCS) u otro LLC y las capas físicas.

El administrador de servicios MAC tiene 26 primitivas. Comparadas con el 802.15.1 (bluetooth), que tiene alrededor de 131 primitivas en 32 eventos, el MAC 802.15.4 es muy simple, haciéndolo muy versátil para las aplicaciones hacia las que fue orientado, aunque se paga el costo de tener un instrumento con características menores a las del 802.15.1.

1.3.1 Formato general de tramas MAC.

El formato general de las tramas MAC fue diseñado para ser bastante flexible y que se ajustara a las demandas de las diferentes aplicaciones con diversas topologías de red al mismo tiempo que se mantiene un protocolo simple.

El formato general de una trama MAC se muestra en la Figura 4. A la trama MAC se le denomina unidad de datos de protocolos MAC (MPDU) y se compone de la cabecera MAC (MHR), unidad de servicio de datos MAC (MSDU) y *footer* de la MAC (MFR).

El primer campo de la cabecera de trama es el campo de control. En él se indica el tipo de trama MAC que se trasmite, especifica el formato y la dirección de campo y controla los mensajes de ACK (reconocimiento).

En resumen, el campo de control especifica como es el resto de la trama de datos y qué es lo que contiene.

El tamaño de las direcciones puede variar entre 0 y 20 bytes. Por ejemplo, una trama de datos puede contener información del origen y del destino, mientras que la trama de ACK no contiene ninguna información de ninguna dirección. Por otro lado una trama de *beacon* puede sólo contener información de la dirección de la fuente. Ésta flexibilidad de la estructura ayuda a incrementar la eficiencia del protocolo para el almacenamiento de paquetes.

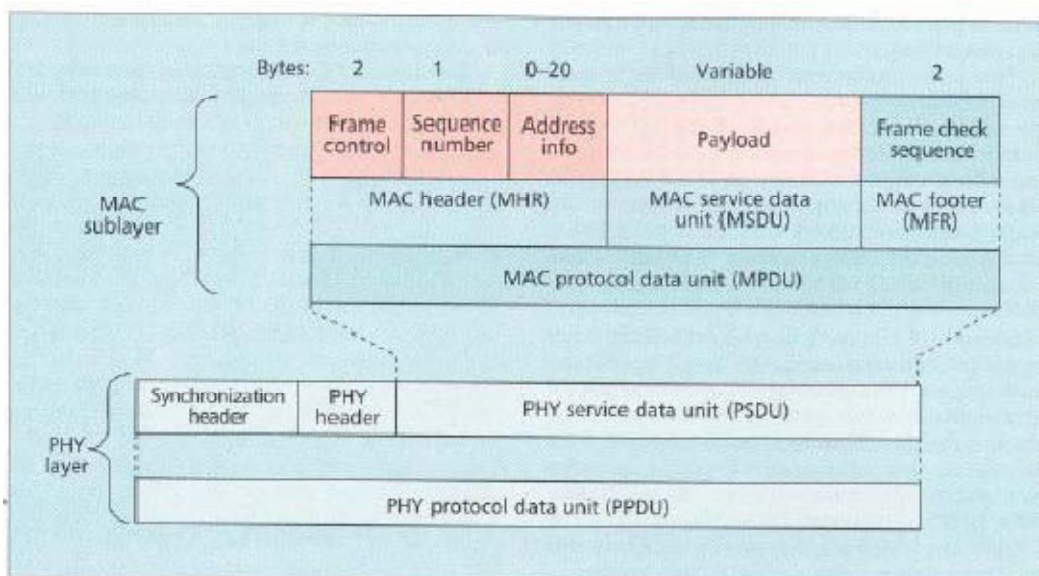


Figura 4 - Formato de trama MAC

El campo payload, que es donde se enviarán los datos, será de longitud variable; sin embargo, la trama completa de MAC no debe de exceder los 127 bytes de información. Los datos que lleva el payload dependerán entonces del tipo de trama.

El estándar IEEE 802.15.4 tiene cuatro diferentes tipos de tramas: trama de *beacon*, de datos, tramas de ACK y tramas de comandos MAC. Solo las tramas de datos y de beacon contienen información proveniente de capas superiores. Las tramas de mensajes de ACK y la de comandos MAC originados en el MAC son usadas para comunicaciones MAC peer-to-peer. Otros campos en la trama MAC son sequence number y tramas de chequeo (FCS). El número de secuencia de la cabecera enlaza a las tramas de ACK con envíos anteriores. La transmisión se considera exitosa solo cuando la trama de ACK tiene el mismo número de secuencia que la trama anterior transmitida. Las FCS ayudan a verificar la integridad de las tramas del MAC.

1.3.2 La estructura de las supertramas.

Algunas aplicaciones requieren anchos de banda dedicados para lograr estados de bajo consumo de potencia. Para lograr dichos estados latentes el IEEE 802.15.4 se puede operar en un modo opcional llamado supertramas (superframes).

En una supertrama un coordinador de red, denominado el coordinador PAN, transmite los beacons de la supertrama en intervalos determinados. Estos intervalos pueden ser tan cortos como unos 15 ms o tan largos como 245 s.

El tiempo entre cada uno de ellos se divide en 16 ranuras de tiempo, o slots, independientes a la duración de cada superframe. Un mote (nodo) puede transmitir cuando sea durante una ranura de tiempo. Pero debe de terminar su transmisión antes del siguiente superframe.

Un *superframe* consiste en un periodo denominado CAP (Contention Access Period) seguido de otro periodo denominado CFP (Contention Free Period) y un periodo inactivo en el cual el coordinador podrá estar en el modo de ahorro de energía y no hará nada.

Un nodo podrá transmitir datos en el periodo CAP empleando el algoritmo CSMA-CA, sin embargo el coordinador PAN puede asignar intervalos o ranuras de tiempo a un solo dispositivo que requiera un determinado ancho de banda permanentemente o transmisiones latentes bajas.

Estas ranuras de tiempo asignadas son llamadas ranuras de tiempo garantizadas (Guarantee Time Slot - GTS) y juntos forman el periodo de contención libre (Contention Free Period). El tamaño del periodo de contención libre puede variar dependiendo de la demanda de los demás dispositivos asociados a la red. Cuando se utiliza el GTS, todos los dispositivos deben de completar todas sus transmisiones de contención antes de que el periodo de contención libre comience. En la Figura 5 podemos ver la estructura de una supertrama y sus distintos tipos de ranuras.

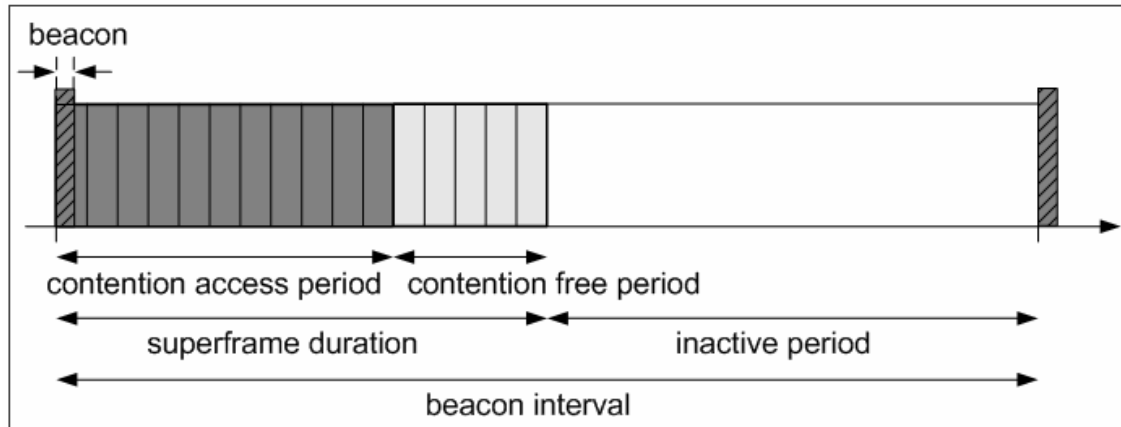


Figura 5 - Estructura de una supertrama

En una red beacon-enabled, se emplea el mecanismo de acceso al canal slotted CSMA/CA, por lo que los dispositivos deben esperar a las tramas de sincronismo para transmitir datos. Cualquier dispositivo que quiera transmitir durante el periodo de contención espera a que empiece la siguiente ranura de tiempo y después determina si algún otro dispositivo se encuentra transmitiendo en la misma ranura de tiempo. Si algún otro dispositivo se encuentra transmitiendo en dicho slot, el dispositivo se repliega a un número aleatorio de slots o indica un fallo en la conexión después de varios intentos. Además en una red beacon-enabled, las tramas de acknowledgment no utilizan CSMA.

En una red non-beacon-enabled se emplea el mecanismo de acceso al canal unslotted CSMA/CA, por lo que los dispositivos no deben esperar a las tramas de sincronismo para transmitir datos. En este tipo de redes, los dispositivos están continuamente activos y a la espera de recibir paquetes, lo que requiere requisitos de potencia mayores.

1.4 Capa física.

El IEEE 802.15.4 provee dos opciones de PHY que combinan con el MAC para permitir un amplio rango de aplicaciones en red. Ambas PHYs se basan en métodos de secuencia directa de espectro extendido (DSSS). Se hace así porque da lugar a bajos costes de implementación digital en circuitos integrados, y ambas comparten la misma estructura básica de paquetes low-duty-cycle con operaciones de bajo consumo de energía. La principal diferencia entre ambas PHYs radica en la banda de frecuencias. La PHY de los 2.4 GHz, especifica operación en la banda industrial, médica y científica (ISM), que prácticamente está disponible a nivel mundial, mientras que la PHY de los 868/915 MHz especifica operaciones en la banda de 865 MHz en Europa y 915 MHz en la banda ISM en Estados Unidos. Mientras que la movilidad entre países no se anticipa para la mayoría de las aplicaciones de redes en viviendas, la disponibilidad internacional de la banda de los 2.4 GHz ofrece ventajas en términos de mercados más amplios y costes de manufactura más bajos. Por otro lado las bandas de 868 MHz y

915 MHz ofrecen una alternativa a la congestión creciente y demás interferencias (hornos de microondas, etc.) asociadas a la banda de 2.4 GHz. También ofrece mayores rangos por enlace debido a que existen menores pérdidas de propagación.

Existe una segunda distinción de las características de la PHY, que es el rango de transmisión. La PHY de 2.4 GHz permite un rango de transmisión de 250 kb/s, mientras que la PHY de los 868/915 MHz ofrece rangos de transmisión de 20 kb/s y 40 kb/s respectivamente. Este rango superior de transmisión en la PHY de los 2.4 GHz se atribuye principalmente a un mayor orden en la modulación, en la cual cada símbolo representa múltiples bits. Los distintos rangos de transmisión se pueden usar para lograr una variedad de desarrollos o aplicaciones. Por ejemplo, al tener baja densidad de datos en la PHY de los 868/915 MHz, se puede ocupar para lograr mayor sensibilidad y mayores áreas de cobertura, con lo que se reduce el número de nodos requeridos para cubrir un área geográfica, mientras que el rango superior de transmisión en la PHY de los 2.4 GHz se puede utilizar para conseguir throughput mayores y poca latencia. Se espera que en cada PHY se encuentren aplicaciones adecuadas a ellas y a sus rangos de transmisión.

1.5.- Canales para transmisión y recepción.

En el IEEE 802.15.4 se definen 27 canales de frecuencia entre las tres bandas. La PHY de los 868/915 MHz soporta un solo canal entre los 868 y los 868.6 MHz, y diez canales entre los 902.0 y 928.0 MHz. Debido al soporte regional de esas dos bandas de frecuencias, es muy improbable que una sola red utilice los 11 canales. Sin embargo, las dos bandas se consideran lo suficientemente cercanas en frecuencia que se puede utilizar el mismo hardware para ambos y así reducir costos de manufactura. La PHY de los 2.4 GHz soporta 16 canales entre los 2.4 y los 2.4835 GHz con un amplio espacio entre canales (5 MHz) esto con el objetivo de facilitar los requerimientos de filtrado en la transmisión y en la recepción. Podemos ver los dos tipos de estructura de los canales en 802.15.4 en la Figura 6, y en la Figura 7 el cálculo de la frecuencia central de cada uno de ellos.

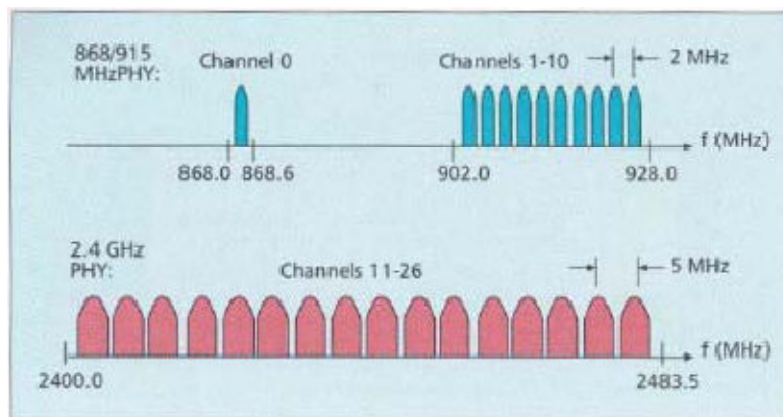


Figura 6 - Estructura de los canales en 802.15.4

Channel number	Channel center frequency (MHz)
$k = 0$	868.3
$k = 1, 2, \dots, 10$	$906 + 2(k - 1)$
$k = 11, 12, \dots, 26$	$2405 + 5(k - 11)$

Figura 7 - Frecuencia de los canales

Debido a que puede haber multitud de interferencias producidas por redes inalámbricas trabajando en la misma banda de frecuencia, o interferencias no intencionadas de otras aplicaciones, el poder relocalizar dentro del espectro nuestra aplicación será un factor muy importante para el éxito de dichas redes inalámbricas.

El estándar fue diseñado para implementar una selección dinámica de canales, a través de una selección específica de algoritmos la cual es responsabilidad de la capa de red. La capa MAC incluye funciones de búsqueda que sigue paso a paso a través de una lista de canales permitidos en busca de un *beacon*, mientras que la PHY contiene varias funciones de bajo nivel, tales como la detección de los niveles de energía recibidos, indicadores de calidad en el enlace así como de conmutación de canales, lo que permite asignación de canales y agilidad en la selección de frecuencias. Esas funciones son utilizadas por la red para establecer su canal inicial de operación y para cambiar canales en respuesta a una pausa muy prolongada.

1.6 Estructura de paquetes de información.

Para mantener una interfaz común y simple con el MAC, ambas capas PHY comparten una estructura simple de paquete. Cada paquete, o unidad de datos del protocolo PHY (PPDU), contiene un encabezado de sincronización, un encabezado de PHY para indicar la longitud del paquete, y la carga de información, o la unidad de secuencia PHY (PSDU). Podemos ver el paquete PHY detallado en la Figura 8. El preámbulo de 32 bits está diseñado para la adquisición de símbolos y para los tiempos de chip, y en algunos casos se utiliza para ajustes bruscos en la frecuencia. No se requiere una ecualización en el canal de la PHY debido a la combinación de áreas de cobertura pequeñas con rangos bajos de transmisión.

Dentro de la cabecera del paquete PHY, se utilizan 7 bits para especificar la longitud de la carga de datos (en bytes). La longitud de paquetes va de 0 a 127 bytes, a través del overhead de la capa MAC, paquetes con longitud cero no ocurren en la práctica.



Figura 8 - Paquete PHY

1.7 Modulación.

La PHY en los 868/915 MHz utiliza una aproximación simple DSSS [4] en la cual cada bit transmitido se representa por un chip-15 de máxima longitud de secuencia (secuencia m). Los datos binarios son codificados al multiplicar cada secuencia m por +1 o -1, y la secuencia de chip que resulta se modula dentro de la portadora utilizando BPSK (binary phase shift keying). Antes de la modulación se utiliza una codificación de datos diferencial para permitir una recepción diferencial coherente de baja complejidad.

La PHY de 2.4 GHz emplea una técnica de modulación semi-ortogonal basada en métodos de DSSS (con propiedades similares). Los datos binarios son agrupados en símbolos de 4 bits, y cada símbolo especifica una de las 16 secuencias de transmisión semi-ortogonales de código de pseudo-ruido (PN). Las secuencias de PN son concatenadas para que sean datos de símbolos exitosos, y la secuencia agregada al chip es modulada en la portadora utilizando MSK (minimum shift keying). El uso de símbolos "casi ortogonales" simplifica la implementación a cambio de un desempeño ligeramente menor (< 0.5 dB). Los parámetros de modulación para ambas PHY se resumen a continuación en la Tabla 2.

PHY	Banda	Parámetro de datos			Parámetros de riesgo	
		Velocidad de bits (Kb/s)	Velocidad de símbolos (kbaud)	Modulación	Velocidad de chip (Mchips/s)	Modulación
868/915 (MHz)	868.0-868.6 (MHz)	20	20	BPSK	0.3	BPSK
	902.0-928.0 (MHz)	40	40	BPSK	0.6	BPSK
2.4 (GHz)	2.4 – 4.4835 (GHz)	250	62.5	16 - semi ortogonal	2.0	O-QPSK

Tabla 2 - Parámetros de modulación

En términos de eficiencia (energía requerida por bit), la señalización ortogonal mejora su desempeño en 2 dB que BPSK diferencial. Sin embargo, en términos de sensibilidad de recepción, la PHY 868/915 PHY tiene una ventaja de 6-8 dB debido a que tiene velocidades de transmisión más bajas. En ambos casos las pérdidas de implementación debido a la sincronización, forma del pulso, simplificaciones en el detector, y demás resultan en desviaciones en sus curvas óptimas de detección.

1.8 Sensitividad y rango.

Las especificaciones actuales de sensibilidad del IEEE 802.15.4 especifican -85 dBm para la PHY de los 2.4 GHz y de -92 dBm para la PHY de los 868-915 MHz.

Dichos valores incluyen suficiente margen para las tolerancias que se requieren debido a las imperfecciones en la manufactura de la misma manera que permite implementar aplicaciones de bajo coste. En cada caso, los dispositivos suelen estar en el orden de 10 dB mejores que las especificaciones.

Naturalmente el rango deseado estará en función de la sensibilidad del receptor así como de la potencia del transmisor. El estándar especifica que cada dispositivo debe ser capaz de transmitir al menos 1 mW, pero dependiendo de las necesidades de la aplicación, la potencia de transmisión puede ser mayor o menor, la potencia actual de transmisión puede ser menor o mayor (dentro de los límites de regulación establecidos). Los dispositivos típicos (1mW) se espera que cubran un rango de entre 10-20 m; sin embargo, con una buena sensibilidad y un incremento moderado en la potencia de transmisión, una red con topología tipo estrella puede proporcionar una cobertura total para una casa. Para aplicaciones que requieran mayor tiempo de latencia, la topología tipo *mesh* (malla) ofrece una alternativa atractiva con coberturas pequeñas dado que cada dispositivo solo necesita suficiente energía para comunicarse con su vecino más cercano.

1.9.- Interferencia de y para otros dispositivos

Los dispositivos que operan en la banda de los 2.4 GHz pueden recibir interferencias causadas por otros servicios que operan en dicha banda. Esta situación es aceptable en las aplicaciones que utilizan el estándar IEEE 802.15.1, las cuales requieren una baja calidad de servicio (QoS), no requiere comunicación asíncrona, y se espera que realice varios intentos para completar la transmisión de paquetes. Por el contrario, un requerimiento primario de las aplicaciones del IEEE 802.15.4 es una larga duración en baterías. Este objetivo se logra con poca energía de transmisión y muy pocos ciclos de servicio. Dado que los dispositivos IEEE 802.15.4 se pasan dormidos el 99 % del tiempo, y emplea transmisiones de baja energía en el espectro ensanchado, deben estar trabajando en las vecindades de la banda de los 2.4 GHz.

2. TINYOS.

2.1 Introducción.

TinyOS es un sistema operativo gratuito y de código abierto basado en componentes, y también una plataforma enfocada a las redes de sensores inalámbricos (WSNs).

Es un sistema operativo embebido programado en el lenguaje nesC (un meta-lenguaje derivado de C) como un set de tareas colaborativas y procesos.

TinyOS surgió como una colaboración entre las Universidades de California, la de Berkeley y en cooperación con Intel Research y la empresa Crossbow Technology, y ha ido creciendo hasta convertirse en un consorcio internacional conocido como la Alianza TinyOS.

2.2 Implementación.

Las aplicaciones TinyOS están escritas en nesC, un dialecto del lenguaje de programación C, optimizado para las limitaciones de memoria de las redes de sensores. Sus aplicaciones suplementarias suelen ser front-ends Java y Shell Script, es decir la parte que interactúa con el usuario. Las librerías asociadas y herramientas como el mismo compilador de NesC, herramientas del Atmel AVR (microcontrolador) están escritas principalmente en lenguaje C.

Los programas TinyOS están construidos a través de componentes de software, algunas de las cuales presentan abstracciones de Hardware. Las componentes se conectan entre sí a través de interfaces. TinyOS provee interfaces y componentes para abstracciones comunes como: comunicación de paquetes, enrutado, medida de sensores, actuación y almacenamiento.

TinyOS es completamente sin bloqueo y tiene una pila única, es decir, permite a otros procesos continuar antes que la transmisión haya concluido. Por consiguiente, todas las operaciones de entrada salida (E/S) que duren algo más que cientos de microsegundos son asíncronas y tienen una función de llamada de regreso.

Para permitir al compilador nativo optimizar a través de la limitación de las llamadas, TinyOS usa las cualidades de nesC para enlazar estas llamadas de retorno, llamadas eventos, estáticamente.

El diseño de TinyOS está basado en responder a las características y necesidades de las redes de sensores, tales como reducido tamaño de memoria, bajo consumo de energía, operaciones de concurrencia intensiva, diversidad en diseños y usos, y

finalmente operaciones robustas para facilitar el desarrollo confiable de aplicaciones. Además se encuentra optimizado en términos de uso de memoria y eficiencia de energía.

El diseño del Kernel de TinyOS está basado en una estructura de dos niveles de planificación.

- **Eventos:** Pensados para realizar un proceso pequeño (por ejemplo cuando el contador del timer se interrumpe, o atender las interrupciones de un conversor analógico-digital). Además pueden interrumpir las tareas que se están ejecutando.
- **Tareas:** Las tareas son pensadas para hacer una cantidad mayor de procesamiento y no son críticas en tiempo (por ejemplo calcular el promedio en un arreglo). Las tareas se ejecutan en su totalidad, pero la solicitud de iniciar una tarea, y el término de ella son funciones separadas. Esta característica es propia de la programación orientada a componentes.

Con este diseño permitimos que los eventos (que son rápidamente ejecutables), puedan ser realizados inmediatamente, pudiendo interrumpir a las tareas (que tienen mayor complejidad en comparación a los eventos).

El enfoque basado en eventos es la solución ideal para alcanzar un alto rendimiento en aplicaciones de concurrencia intensiva. Adicionalmente, este enfoque usa las capacidades de la CPU de manera eficiente y de esta forma gasta el mínimo de energía.

2.2.1 nesC.

NesC es un meta-lenguaje de programación basado en C, orientado a sistemas embebidos que incorporan el manejo de red. Además soporta un modelo de programación que integra el manejo de comunicaciones, las concurrencias que provocan las tareas y eventos y la capacidad de reaccionar frente a sucesos que puedan ocurrir en los ambientes donde se desempeña (por ejemplo, medir de un sensor).

También realiza optimizaciones en la compilación del programa, detectando posibles datos en ejecución que pueden ocurrir producto de modificaciones concurrentes a un mismo estado, dentro del proceso de ejecución de la aplicación. Además simplifica el desarrollo de aplicaciones, reduce el tamaño del código, y elimina muchas fuentes potenciales de errores.

NesC nos ofrece:

- Separación entre la construcción y la composición. Hay dos tipos de componentes en NesC: módulos y configuraciones. Los módulos proveen el código de la aplicación, implementando una o más interfaces. Estas interfaces son los únicos puntos de acceso a la componente. Las configuraciones son usadas para unir las componentes entre sí, conectando las interfaces que algunas componentes proveen con las interfaces que otras usan.
- Interfaces bidireccionales: tal como ya se explicaba anteriormente las interfaces son los accesos a las componentes, conteniendo comandos y eventos, los cuales son los que implementan las funciones. El proveedor de una interfaz implementa los comandos, mientras que el que las utiliza implementa eventos.
- Unión estática de componentes, vía sus interfaces. Esto aumenta la eficiencia en tiempos de ejecución, incrementa la robustez del diseño, y permite un mejor análisis del programa.
- NES C posee herramientas que optimizan la generación de códigos. Un ejemplo de esto es el detector de datos en ejecución en tiempo de compilación.

Como resumen, los principales aspectos que el modelo de programación NesC ofrece y que deben ser entendidos para el entendimiento del diseño con TinyOS son:

- el modelo de NesC está formado por interfaces y componentes.
- Una interfaz puede ser usada o puede ser provista, las componentes son módulos o configuraciones.
- Una aplicación se verá representada como un conjunto de componentes, agrupados y relacionados entre sí, tal como se observa en la figura.

La Figura 9 nos muestra un ejemplo de aplicación programa en nesC con varias componentes de software.

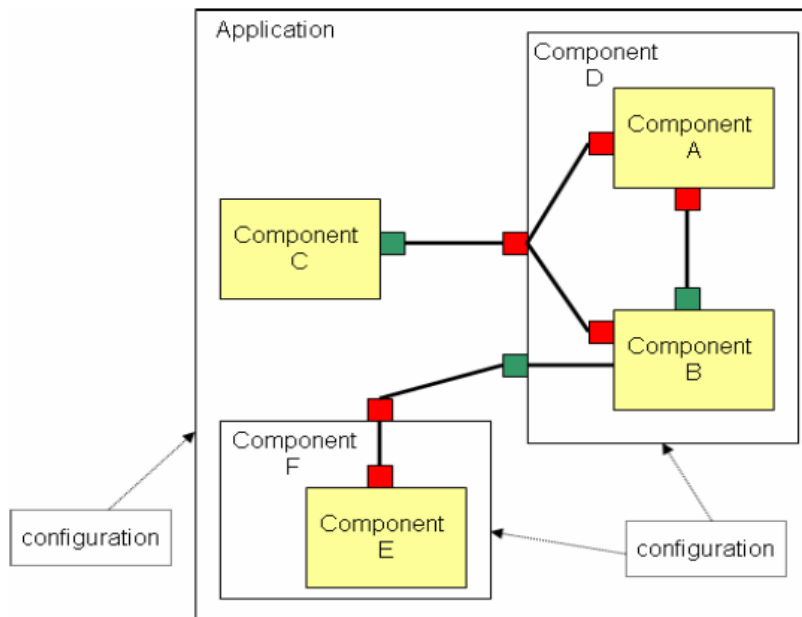


Figura 9 - Ejemplo de aplicación con varias componentes

- Las interfaces se utilizan para operaciones que describen la interacción bidireccional; el proveedor de la interfaz debe implementar comandos, mientras que el usuario de la interfaz debe implementar eventos.
- Existen dos tipos de componentes:
 - Módulos, que implementan especificaciones de una componente.
 - Configuraciones, que se encargarán de unir (*wire*) diferentes componentes en función de sus interfaces, ya sean comandos o eventos.

2.3 Tutoriales de TinyOS

En la página web de TinyOS podremos encontrar los links a un wiki con una serie de tutoriales con los cuales lograremos un buen conocimiento de TinyOS, el entorno de programación, manejo de sensores, y algunas de las particularidades del sistema operativo y de nesC:

- Conceptos principales de TinyOS: componentes, módulos, configuraciones e interfaces. Nos enseñara también a compilar e instalar un programa TinyOS en un mote.
- Modelo de ejecución de TinyOS y el trabajo con módulos. Eventos, comandos y su relación con las interfaces en mayor profundidad, introduciendo las operaciones de fase partida. Explicación de qué son las tareas, el mecanismo en TinyOS para que los componentes compartan el procesador de un modo cooperativo.

- Modelo principal de comunicación de TinyOS. Comunicación nodo a nodo. Ejercicios tipo de envío y recepción de mensajes para ver cómo funciona una red.
- Modelo de comunicación entre un nodo y el PC. Introducción a las herramientas de comunicación para PC's y portátiles para comunicarse con los nodos a través del puerto serie. Conceptos básicos de originador de paquetes. Explicación de la herramienta Mig, y del SerialForwarder.
- Lectura de sensores en TinyOS. Ejemplos de muestreo de sensores en nodos y representación de los valores o resultados en leds. Explica cómo se recogen los datos de los distintos sensores que encontraremos en los distintos nodos que tengamos en nuestra red.
- Secuencia de arranque de programas. Detallado de cómo iniciar un programa en TinyOS y respuestas acerca de dónde se encuentra main() dentro del código de un programa escrito en nesC.
- Almacenamiento. Se explica la forma de almacenar datos en TinyOS de forma permanente (no volátil). Aplicaciones ejemplo detallaran el uso de las interfaces Mount, ConfigStorage, LogRead y LogWrite. Se aprenderá a dividir el chip de la memoria flash en distintos volúmenes, lo cual permite almacenar múltiples y/o distintos tipos de datos.
- Administración o métodos de arbitraje de recursos. Modelo de manejo de energía. Se enseña cómo ganar el acceso a recursos compartidos de manera predefinida, y cómo crear nuestros propios recursos compartidos en TinyOS. Presentación para controlar los estados de energía en ambos casos. Virtualización.
- Plataformas. Comprensión de las distinciones entre las múltiples plataformas de motes, incluyendo como esto diferencia en ficheros, directorios, y definiciones a usar. Guía de iniciación para comprender mejor el sistema make o para diseñar una nueva plataforma dentro de TinyOS.

2.4 Open-Zigbee.

Open Zigbee es un software (código fuente) de código abierto y gratuito que desarrolla la pila del protocolo IEEE 802.15.4/Zigbee dentro de TinyOS. En particular la capa física (PHY) y la MAC.

El proyecto está siendo desarrollado con la colaboración de IPP-Hurray y ART-WiSE framework.

Esta implementación está escrita en nesC y la razón para la creación de esta tecnología estándar es el lograr una mayor difusión de las WSN (Wireless Sensor Networks). La implementación de código abierto de 802.15.14 además nos permite el testeo, implementación y validación de los mecanismos que posee 802.15.4 junto con nuestros propios proyectos y aplicaciones. Para conseguir esa meta se crea un entorno de trabajo y la implementación del protocolo en TinyOS.

Además, no existe ninguna otra implementación del protocolo Zigbee dentro de TinyOS. Es por ello que el proyecto de código abierto Open-Zigbee trabaja junto con TinyOS para que en la versión 2.x de TinyOS venga de facto una implementación robusta de Zigbee.

La implementación de Open-Zigbee es soportada por el hardware de las plataformas MicaZ y TelosB.

2.4.1 Funcionalidades de Open-Zigbee.

Gracias a la implementación de código abierto se han desarrollado diversas funciones aparte de las que provee 802.15.4. Éstas son algunas de las funcionalidades que se han desarrollado con Open-Zigbee:

- Implementación del algoritmo CSMA/CA slotted.
- Mecanismo de Asociación y Disociación.
- Gestión de Beacon (mecanismo de balizas).
- Mecanismo de reserva de slot garantizado (GTS).
- Transmisión indirecta/ Datos Pendientes.
- Escaneo de canal.
- Construcción de trama.
- Temporizadores de eventos de MAC.
- TimerAsync y sincronización.

2.4.2 Estructura de la implementación.

Este esquema nos sirve para comprender como está organizada la pila de protocolo 802.15.4 en Open Zigbee.

La implementación reutiliza ficheros del árbol de directorio de TinyOS, por lo que se debe instalar en contrib/hurray. La estructura de directorios de Open Zigbee es similar a la que tenemos en TinyOS 2.x, y la podemos ver en la Figura 10 a continuación:

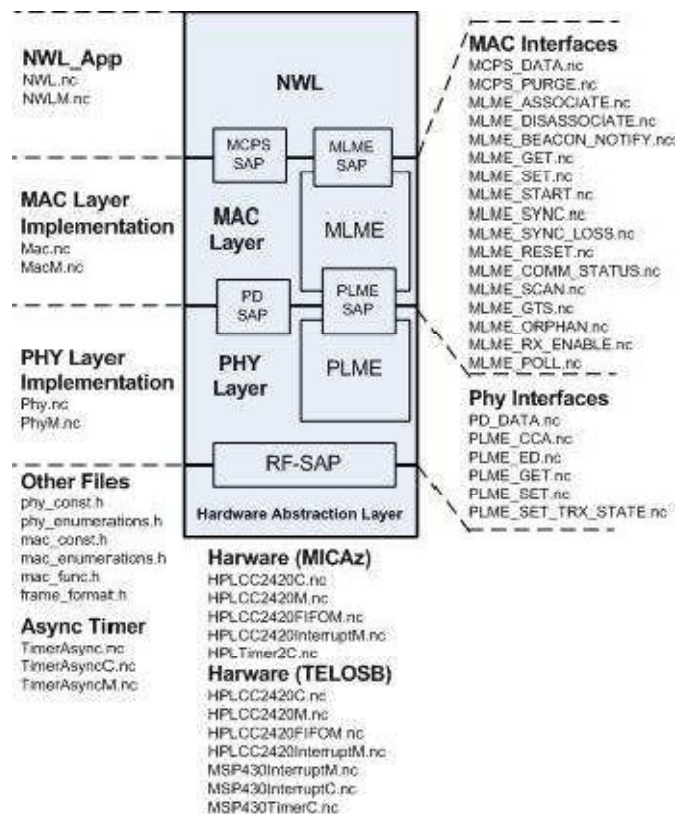


Figura 10 - Arquitectura de la pila del protocolo

La siguiente lista representa los ficheros creados para ésta implementación y su localización:

contrib/hurray/tos/interfaces/ieee802154.mac – Interfaces de conexión entre la MAC y las capas superiores.

- `MCPS_DATA.nc` – MAC Parte común de la subcapa dato-servicio del punto de acceso.
- `MCPS_PURGE.nc` - MAC Parte común de la subcapa de inicio de punto de acceso.
- `MLME_ASSOCIATE.nc` – Entidad de manejo de la Asociación de Servicio de la capa MAC.
- `MLME_BEACON_NOTIFY.nc` – Administrador de capa MAC para las señales de notificación del servicio de punto de acceso.
- `MLME_COMM_STATUS.nc` – Entidad de manejo de la capa MAC encargada del servicio de punto de acceso al estado de la comunicación.
- `MLME_DISASSOCIATE.nc` –Entidad de manejo de la capa MAC encargada del servicio de punto de acceso a la disociación.
- `MLME_GET.nc` – Entidad de manejo de la capa MAC encargada del servicio de la toma de punto de acceso.
- `MLME_GTS.nc` – Entidad de manejo de la capa MAC encargada del servicio de punto de acceso a GTS.
- `MLME_POLL.nc` – Entidad de manejo de la capa MAC encargada del servicio de punto de acceso a encuesta.
- `MLME_RESET.nc` – Entidad de servicio de la capa MAC encargada del servicio de punto de acceso a Reset.

- MLME_SCAN.nc - Entidad de servicio de la capa MAC encargada del servicio de punto de acceso a "Scan".
- MLME_SET.nc - Entidad de servicio de la capa MAC encargada del servicio de punto de acceso a "Set".
- MLME_START.nc - Entidad de servicio de la capa MAC encargada del servicio de punto de acceso a "Start".
- MLME_SYNC.nc - Entidad de servicio de la capa MAC encargada del servicio de punto de acceso a "Sync" (sincronización)
- MLME_SYNC_LOSS.nc - Entidad de servicio de la capa MAC encargada del servicio de punto de acceso a "Sync Loss" (pérdida de la sincronización).

contrib/hurray/tos/interfaces/ieee802154.phy – Interfaces de conexión entre la capa MAC y las capas PHY.

- PD_DATA.nc – PHY Punto de acceso de Dato-Servicio.
- PLME_CCA.nc – Entidad de manejo de la capa física – Valoración de liberación del canal- Servicio de punto de acceso.
- PLME_ED.nc – Entidad de manejo de la capa física encargada del servicio de punto de acceso de detección de energía.
- PLME_GET.nc - Entidad de manejo de la capa física encargada del servicio de punto de acceso de "Get".
- PLME_SET.nc - Entidad de manejo de la capa física encargada del servicio de punto de acceso de "Set"
- PLME_SET_TRX_STATE.nc – Entidad de manejo de la capa física encargada del acceso al servicio-estado de "Set Transceiver".

contrib/hurray/tos/lib/mac – Ficheros de implementación de la capa MAC.

- Mac.nc – Configuración del modulo de implementación MacM.
- MacM.nc – Implementación de la capa MAC.
- mac_const.h – Constantes de la capa MAC.
- mac_enumerations.h – Enumeraciones de la capa MAC.

contrib/hurray/tos/lib/phy_micaz – Ficheros de implementación de la capa física para la plataforma MICAZ.

- Phy.nc – Configuración del módulo de implementación PhyM.
- PhyM.nc – Implementación de la capa física.
- phy_const.h – Constantes de la capa física.
- phy_enumerations.h – Enumeraciones de la capa física.
- TimerAsync.nc – Interfaz para la componente TimerAsyncM.
- TimerAsyncC.nc – Configuración del modulo de implementación TimerAsyncM.
- TimerAsyncM.nc – Implementación del temporizador asíncrono.

contrib/hurray/tos/lib/phy_telosb – Ficheros de implementación de la capa física para la plataforma TelosB.

- Phy.nc – Configuración del modulo de implementación PhyM.
- PhyM.nc – Implementación de la capa física.
- phy_const.h – Constantes de la capa física.
- phy_enumerations.h – Enumeraciones de la capa física.
- TimerAsync.nc – Interfaz para la componente TimerAsyncM.
- TimerAsyncC.nc – Configuration del modulo de implementación TimerAsyncM.
- TimerAsyncM.nc – Implementación de temporizador asíncrono.

contrib/hurray/tos/system – Módulos genéricos del sistema.

- mac_func.h – Funciones genéricas usadas principalmente por la implementación de la capa MAC.
- frame_format.h – Definición del format de trama.

2.4.3 Implementación de la capa física y MAC

2.4.3.1 Implementación de la capa física

La capa física de IEEE 802.15.4 es responsable de la implementación de las siguientes funcionalidades:

- Activación y desactivación del transmisor de radio;
- Detección de energía dentro del canal actual;
- Indicador de calidad del enlace (LQI) para los paquetes recibidos;
- Clear Channel Assesment (CCA) para Carrier Sense Multiple Access – Collision Avoidance (CSMA-CA);
- Selección de canal de frecuencia;
- Transmisión y recepción de datos;

Tal como podemos ver en la Figura 11, éstas son sus interfaces características.

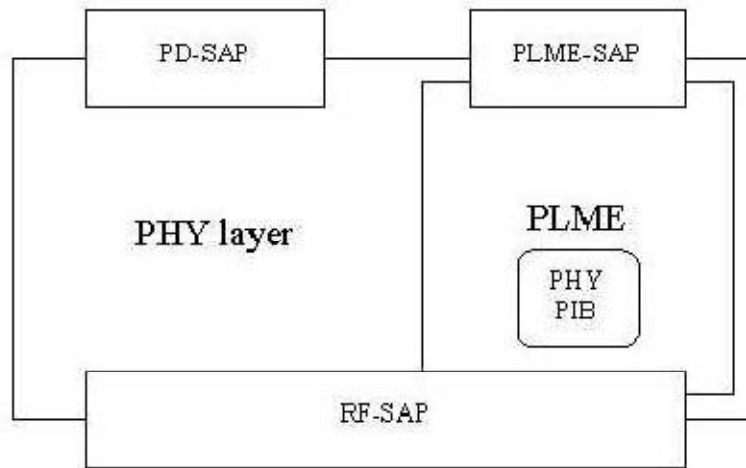


Figura 11 - Modelo de referencia de la capa física

- RF-SAP comprende la interfaz con la radio física a través del firmware de radio frecuencia (RF) del CC2420 (chip de radio de los motes) y Hardware, ya provisto también en la implementación de TinyOS
- El PD-SAP es la interfaz que intercambia paquetes de datos entre las capas MAC y PHY (física).
- PLME-SAP es la interfaz encargada para intercambiar información de administración entre las capas Mac y física.

2.4.3.2 Implementación de la capa MAC

La capa MAC de IEEE 802.15.4 se encarga de la implementación de las siguientes funcionalidades:

- Generación de beacons (balizas) si el dispositivo es un Coordinador.
- Sincronización a los beacons.
- Soporte de asociación y disociación en la PAN (Personal Area Network)
- Soporte para proveer de seguridad al dispositivo.
- Empleo de mecanismo de CSMA/CA para acceso al canal.
- Manejo y mantenimiento del mecanismo GTS (Guarantee Time Slot)
- Provee un link fiable entre dos pares de entidades MAC.

En la Figura 12 podemos ver sus interfaces características.

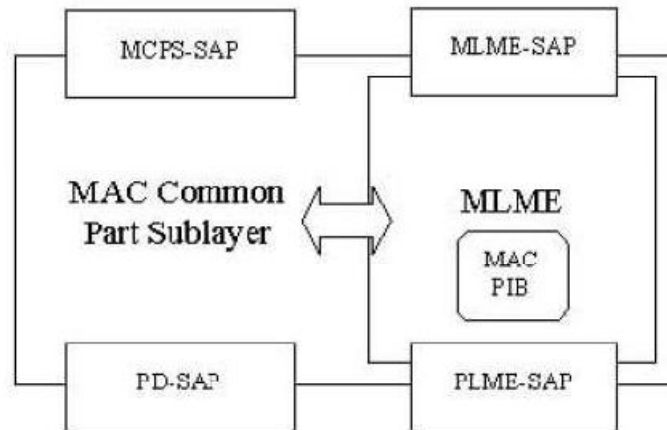


Figura 12 - Modelo de referencia de la capa MAC

La capa MAC provee a las capas superiores dos SAP. El MCPS-SAP (MAC Common Part Sublayer) y el MLME-SAP (MAC Layer Management Entity).

El PD-SAP y el PLME-SAP son usados para conectar la capa MAC con las funcionalidades que provee la capa PHY.

El MCPS-SAP comprende la transferencia de datos MSDU (Mac Service Data Unit) entre la capa MAC y las capas superiores.

El MLME-SAP comprende el intercambio de comandos de manejo entre la capa MAC y las capas superiores.

El MAC PIB (MAC PAN Information Base) es mantenido por la capa MAC y es una base de datos para sus objetos manejados. Las interfaces de MLME-SAP son usadas por la capa superior de MAC para manejar esta información. El PIB almacena la siguiente información:

- Tiempo de espera de ACK;
- Permiso de Asociación;
- Petición automática de Datos;
- Opción de extensión de vida de batería;
- Periodos de extensión de vida de batería;
- Payload de Beacon;
- Tamaño de Beacon Payload;
- Orden de Beacon;
- Tiempo de transmisión de Beacon;
- Número de secuencia de Beacon;
- Dirección extendida del Coordinador;
- Dirección corta del Coordinador;
- Numero de secuencia de datos;
- Opciones de permisos de GTS;
- Número máximo de intentos de retroceso CSMA;

- Exponente mínimo de retroceso;
- Identificador de PAN;
- Opción de Modo Promiscuo;
- Modo de recepción cuando el transceiver este en la opción “libre”;
- Dirección corta;
- Orden de Supertrama;
- Tiempo de persistencia de transacción;

2.4.4 Open-Zigbee 2.0

Para el trabajo del proyecto sobre Open-Zigbee hemos hecho uso de la última implementación beta de Open-Zigbee (hurray2x) que trabaja sobre TinyOS 2.x Para ello se deberán copiar los ficheros de la distribución en:

```
\opt\tinyos-2.x-contrib\
```

3. Tecnología.

Pasamos a comentar la tecnología empleada en este Proyecto Fin de Carrera,

3.1 TelosB.

Los motes TelosB son dispositivos inalámbricos que nos permiten por sus características muchas funcionalidades. Entre ellas están la medición de parámetros en sus sensores, control y seguridad, y otras muchas aplicaciones de las redes de sensores. Podemos ver en la Figura 13 un diagrama de su estructura.

Según las especificaciones los motes TelosB permiten una tasa de transferencia de datos por medio del chip de radio de 250kbps.

Los motes TelosB son fácilmente programables gracias a su interfaz USB. Incorporan en su interior un procesador TI MSP430 que opera a 8MHz y una RAM de 10kbytes. La memoria flash para los programas compilados es de 48kbytes.

Y posee también una memoria flash de 1Mbyte para almacenamiento de aplicaciones. Trabaja en la banda de frecuencia de 2.4 GHz a 2.483Ghz.

Posee una placa de sensores opcional que incorpora: sensores de luz, temperatura y humedad.

Nos centraremos para nuestro Proyecto Fin de Carrera en el sensor de temperatura y humedad SHT-11 (Sensirion SHT-11) que viene en el mote TelosB de manera opcional.

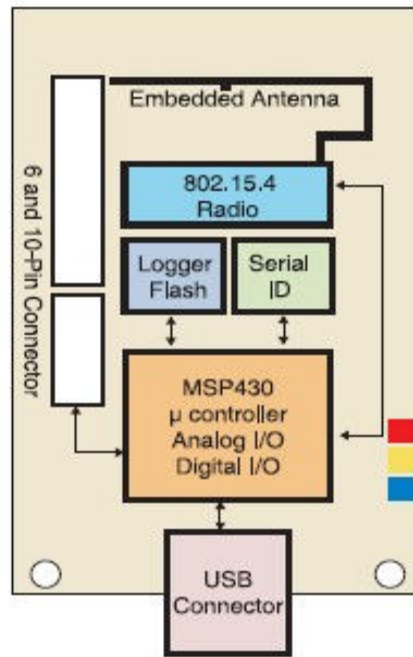


Figura 13 - Diagrama del dispositivo TelosB

3.2 Sensirion SHT-11

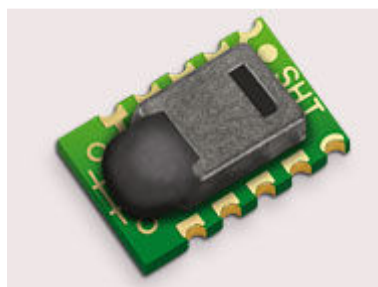


Figura 14 - Sensor Temperatura/Humedad SHT-11

El sensor de temperatura que dispone el mote TelosB está fabricado por la empresa Sensirion, la cual tiene bastante experiencia en la fabricación de sensores para dispositivos. Disponemos de una hoja de especificaciones para comprender los datos de salida del sensor.

Es un sensor con encapsulado SMD (surface mount device), totalmente calibrado y con bajo consumo de energía, y cuya salida es digital. Podemos observarlo en la Figura 14.

La resolución del sensor es de 14bits para temperatura y 12 bits para la humedad.

Para calcular la temperatura en grados centígrados poseemos una tabla de conversión del valor dado por el sensor (a continuación en la Tabla 3), siendo d1 y d2 constantes dependientes del voltaje y los bits de resolución, y SO_T la señal de salida del sensor.

$$T = d_1 + d_2 \cdot SO_T$$

VDD	d ₁ (°C)	d ₁ (°F)	SO _T	d ₂ (°C)	d ₂ (°F)
5V	-40.1	-40.2	14bit	0.01	0.018
4V	-39.8	-39.6	12bit	0.04	0.072
3.5V	-39.7	-39.5			
3V	-39.6	-39.3			
2.5V	-39.4	-38.9			

Tabla 3 - Tabla de conversión de temperatura SHT-11

También disponemos de una fórmula (en la Figura 15) y una gráfica de conversión de los datos de humedad del sensor en humedad relativa (en la Figura 16), donde c1, c2 y c3 son constantes dependientes del número de bits de resolución de salida del sensor (8 o 12).

Notar que para el cálculo de la humedad relativa no depende el voltaje de entrada del sensor.

$$RH_{\text{linear}} = c_1 + c_2 \cdot SO_{RH} + c_3 \cdot SO_{RH}^2 \text{ (%RH)}$$

Figura 15 - Fórmula para el cálculo de la humedad relativa

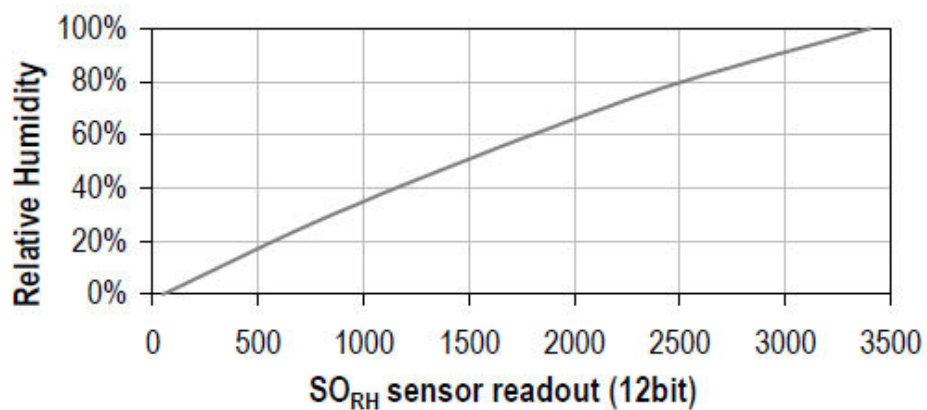


Figura 16 - Gráfica de conversión de Humedad Relativa

3.2.1 Sensirion SHT-11 dentro de TinyOS

Dentro de TinyOS disponemos de dos carpetas con los archivos para el manejo del sensor de temperatura-humedad por parte de alguna aplicación.

Se encuentran en:

`\tos\chips\sht11`

`\tos\platforms\telosb\chips\sht11\`

Mientras que la primera es genérica y multiplataforma, la segunda es específica para los sensores TelosB.

Dentro de la segunda encontraremos:

- `SensirionSht11C` – Es el componente de alto nivel y de presentación del sensor. Las llamadas a la interfaz de lectura de datos del sensor se deberán hacer a ésta componente.
- `HplSensirionSht11C` – Es una componente de bajo nivel, cuya función es proveer los recursos físicos usados por el sensor `Sensirion SHT11` en la plataforma `TelosB`, para que el driver del chip pueda hacer uso de ellos, tales como reloj, datos, pins de energía, etc.
- `HplSensirionSht11P` – Es una componente que controla la energía suministrada al sensor `SHT11` en la plataforma `TelosB`.
- `HalSensirionSht11C` – Es una componente avanzada de acceso para el sensor `SHT11`, en la plataforma `TelosB`. Esta componente provee la interfaz `SensirionSht11`, que ofrece control total sobre el dispositivo.

Dentro de la primera encontraremos:

- `SensirionSht11` – Ésta es la interfaz visible del sensor de temperatura y humedad `SHT11`.
- `SensirionSht11LogicP` – Contiene la lógica actual del driver necesario para leer del sensor `SHT11`. Provee además la interfaz de nivel HAL de `SensirionSht11`.
- `SensirionSht11ReaderP` – Transforma la interfaz HAL de `Sensirion SHT11` en un par de interfaces de lectura SID (Source Independent Driver), una para el sensor de temperatura y la otra para el de humedad.

Veremos a continuación un breve resumen de la abstracción del software en TinyOS 2.x

3.2.2 Abstracción del Hardware. HPL, HAL y HIL.

La introducción de la abstracción de hardware en los sistemas operativos ha demostrado ser valiosa para aumentar la portabilidad y simplificar el desarrollo de aplicaciones, ocultando la complejidad del hardware para el resto del sistema. Sin embargo, las abstracciones de hardware entran en conflicto con el rendimiento y requisitos de eficiencia energética de las aplicaciones de redes de sensores.

Esto lleva a la necesidad de una arquitectura bien definida de las abstracciones de hardware que pueden lograr un equilibrio entre estos objetivos en conflicto. El principal desafío consiste en seleccionar los niveles adecuados de abstracción y organizarlos en forma de componentes TinyOS para apoyar la reutilización, mientras

que mantenemos la eficiencia energética mediante el acceso a las capacidades del hardware completo cuando sea necesario.

En la arquitectura de la figura, la funcionalidad de abstracción de hardware está organizada en tres capas distintas de componentes. Cada capa tiene distintas responsabilidades y depende de las interfaces proporcionadas por las capas inferiores. Las capacidades del hardware subyacente están adaptadas gradualmente a la interfaz establecida (independiente de cualquier plataforma) entre el sistema operativo y las aplicaciones. A medida que avanzamos desde el hardware hacia las interfaces superiores, los componentes se hacen cada vez menos dependientes del hardware, dando al desarrollador más libertad en el diseño y la implementación de aplicaciones reutilizables.

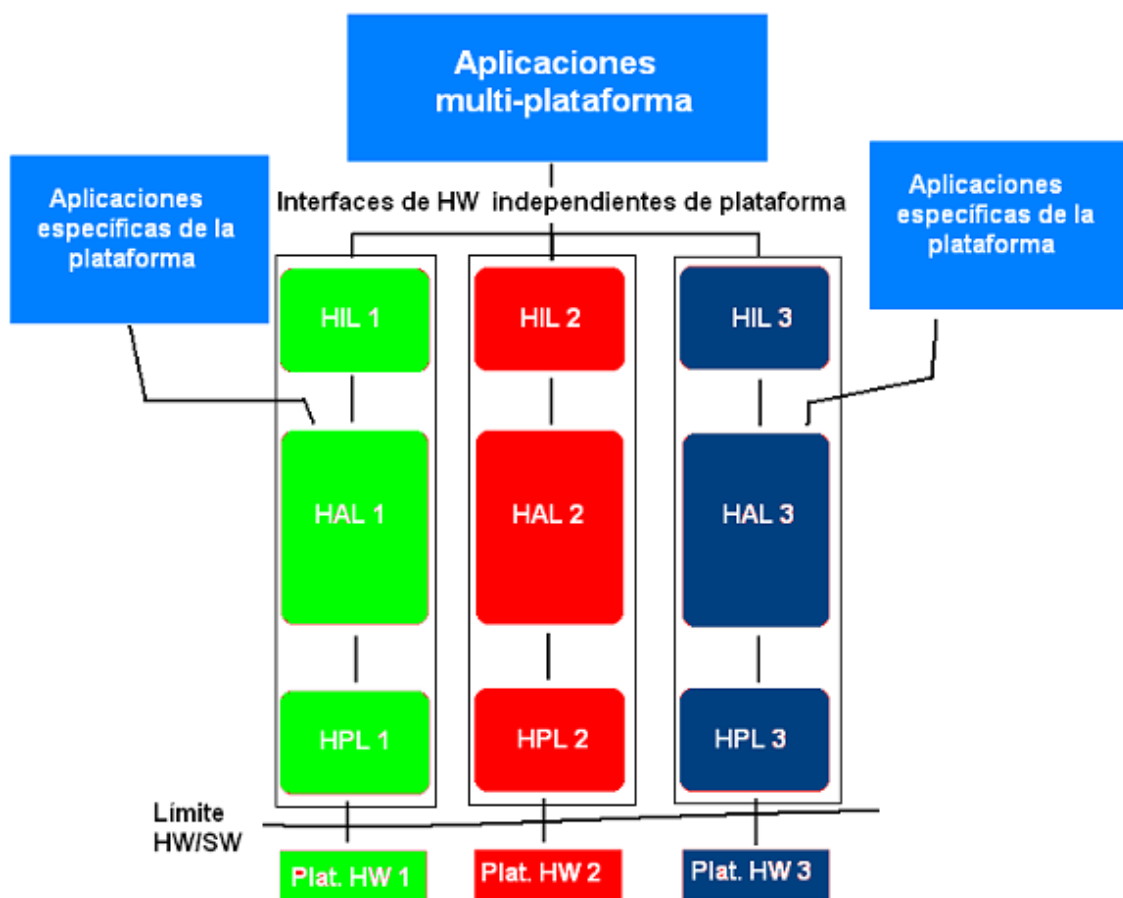


Figura 17 - Arquitectura de la Abstracción del Hardware propuesta

- **HPL (Hardware Presentation Layer – Capa de presentación de Hardware):**

Los componentes que pertenecen a la HPL se colocan directamente sobre el HW / SW de interfaz. Su tarea principal consiste en "presentar" la capacidad del hardware,

utilizando los conceptos nativos del sistema operativo. Ellos acceden al hardware de la forma habitual, ya sea por la memoria o por el puerto asignado E/S. En sentido inverso, el hardware puede solicitar servicio señalando una interrupción.

Usando éstos canales de comunicación internos, el HPL oculta las complejidades de hardware y exporta una interfaz más fácil de leer (llamadas simples a funciones) para el resto del sistema.

- **HAL (Hardware Adaptation Layer – Capa de adaptación de Hardware)**

Los componentes de la capa de adaptación constituyen el núcleo de la arquitectura. Usan las interfaces puras proporcionadas por las componentes de HPL para construir abstracciones útiles ocultando la complejidad naturalmente asociada con el uso de los recursos de hardware. En comparación con los componentes de HPL, les permite mantener el estado que puede ser usado para llevar a cabo el arbitraje y control de recursos.

Debido a los requisitos de eficiencia de las redes de sensores, las abstracciones a nivel de HAL se adaptan al tipo de dispositivo concreto y su plataforma. En vez de ocultar las características individuales de la clase de hardware detrás de modelos genéricos, las interfaces de HAL exponen las características específicas y proporcionan la "mejor" abstracción posible que agiliza el desarrollo de aplicaciones manteniendo al mismo tiempo un uso eficaz de los recursos.

- **HIL (Hardware Interface Layer – Capa de Interfaz de Hardware)**

El nivel final en la arquitectura está formado por los componentes de HIL, que toman las abstracciones (específicas de una plataforma) proporcionadas por la HAL y las convierte a las interfaces de hardware independientes utilizadas por las aplicaciones multiplataforma. Estas interfaces proporcionan una abstracción independiente de la plataforma sobre el hardware que simplifica el desarrollo del software de aplicación ocultando las diferencias de hardware. Para conseguirlo, éste "contrato" de API debería reflejar los servicios típicos de hardware que son necesarios en una aplicación de redes de sensores.

3. IMPLEMENTACIÓN RED DE SENSORES: MUESTRAS DE TEMPERATURA

En éste apartado se va explicar el desarrollo llevado a cabo para la implementación de una red de sensores (WSN) que extraiga datos de un sensor de temperatura en uno de sus nodos y sean transmitidos a otro nodo, cuya conexión al PC nos permita extraer los datos por pantalla.

1. Desarrollo sobre TinyOS 2.0.2

En un principio se opta por trabajar con el sistema operativo TinyOS 2.0.2, bajo el entorno de trabajo Cygwin, y utilizamos como nodos sensores de nuestra red inalámbrica los motes TelosB, por incorporar los mismos el sensor de temperatura y humedad Sensirion SHT-11 anteriormente explicado.

Cygwin es una plataforma bajo Windows que emula el trabajo bajo un entorno UNIX. Podríamos asimismo trabajar directamente sobre una distribución Linux. La más recomendable para la programación de aplicaciones y manejo de redes de sensores inalámbricos (WSN) ya que lleva preinstalado TinyOS es XubunTOS.

El objetivo es crear una red de sensores en la cual un nodo de la red tome muestras del sensor de temperatura que posee y envíe dichos datos a otro nodo inalámbrico que estará conectado a un PC a través del puerto USB. Dicho nodo a través del puerto serie enviará los datos recibidos al PC y mediante una aplicación Java podrán ser mostrados por pantalla con una interfaz gráfica.

La comunicación será bidireccional, puesto que también se pueden enviar datos del nodo conectado al PC al nodo que toma muestras para variar su velocidad de muestreo.

El sistema que se va a desarrollar corresponde a lo representado en la siguiente figura:



Figura 18 - Esquema de desarrollo de la aplicación

Se han implementado dos programas en nesC, que forman nuestro entorno cliente-servidor, que una vez compilados se cargan en los dos motes TelosB:

- Uno de ellos, al cual llamamos "Oscilloscope", tomará muestras y las enviará al otro nodo de la red.
- Y la otra aplicación, "BaseStation" recibe dichas muestras y las envía al puerto serie del PC. Asimismo también enviará datos al nodo sensor para modificar aspectos de su configuración si es necesario.

1.1 BaseStation

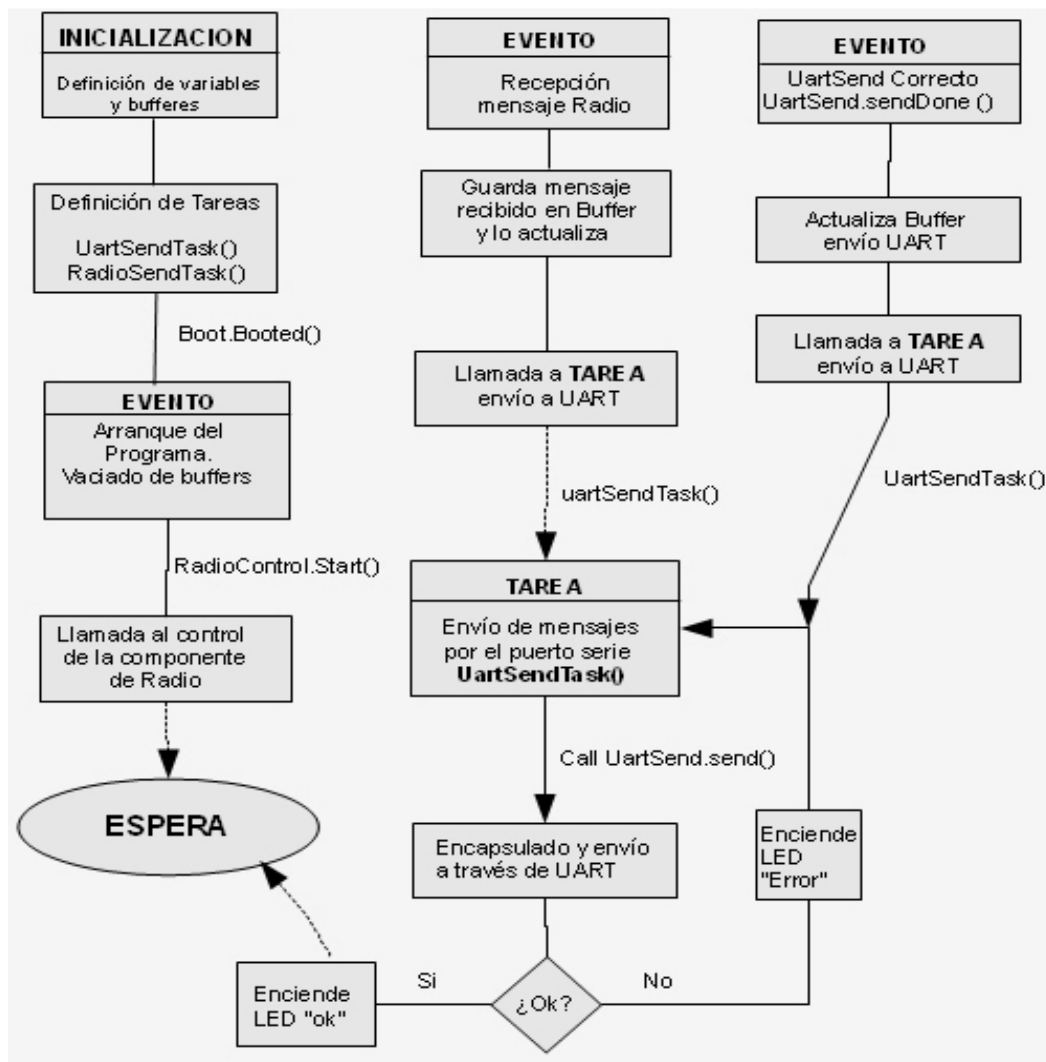
BaseStation será el programa encargado de la recepción de los datos desde el otro nodo dentro de nuestra red y el posterior envío a través del puerto serie al PC.

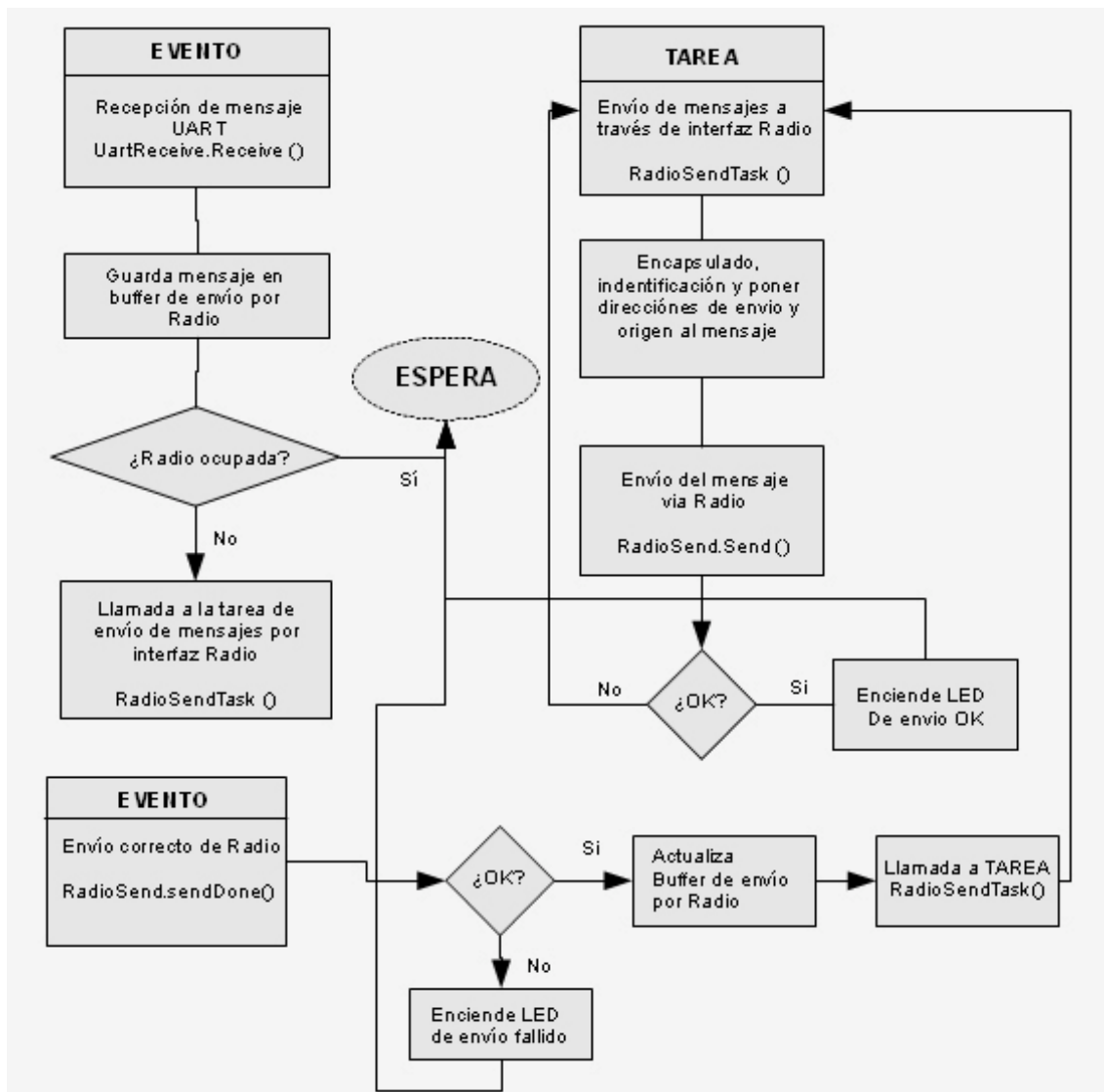
Para evitar que dichos datos enviados desde otro sensor a nuestra estación base o desde la estación base a algún sensor se pierdan existen búferes de transmisión donde se almacenarán los mensajes a enviar.

Los buffers de transmisión via UART o Radio almacenan mensajes del tipo *message_t* y el tamaño de almacenamiento por defecto es de 12 mensajes cada uno de los dos buffers.

Los mensajes enviados en nuestra red son estructuras de datos que poseen los datos a enviar como payload y la respectiva cabecera y pie de mensaje. Además para el envío de mensajes en TinyOS se sigue el modelo de Active Messages, donde cada paquete dentro de la red posee un ID (identificador) para diferenciar dentro de una red las distintas aplicaciones que puedan ocupar el medio.

Diagrama de bloques de BaseStation:





Implementación:

Los ficheros correspondientes a esta implementación se encuentran en el directorio \apps\BaseStation del directorio raíz.

La implementación se desarrolla en el fichero BaseStationP.nc.

Primero se cargarán las interfaces necesarias para el control del dispositivo TelosB

```

module BaseStationP @safe() {
  uses {
    interface Boot;
    interface SplitControl as SerialControl;
  }
}
  
```

```
interface SplitControl as RadioControl;
interface AMSend as UartSend[am_id_t id];

interface Receive as UartReceive[am_id_t id];

interface Packet as UartPacket;

interface AMPacket as UartAMPacket;

interface AMSend as RadioSend[am_id_t id];
interface Receive as RadioReceive[am_id_t id];
interface Receive as RadioSnoop[am_id_t id];
interface Packet as RadioPacket;
interface AMPacket as RadioAMPacket;

interface Leds;
}
}
```

A continuación se definen las variables que usaremos en nuestra aplicación y se inicializan

```
implementation
{
enum {
    UART_QUEUE_LEN = 12,
    RADIO_QUEUE_LEN = 12,
};

message_t  uartQueueBufs[UART_QUEUE_LEN];
message_t  * ONE_NOK uartQueue[UART_QUEUE_LEN];
uint8_t    uartIn, uartOut;
bool       uartBusy, uartFull;

message_t  radioQueueBufs[RADIO_QUEUE_LEN];
message_t  * ONE_NOK radioQueue[RADIO_QUEUE_LEN];
uint8_t    radioIn, radioOut;
bool       radioBusy, radioFull;

task void  uartSendTask();
task void  radioSendTask();
}
```

Creamos llamadas a funciones para el encendido de los Leds del dispositivo

```
void dropBlink() {
    call Leds.led2Toggle();
}

void failBlink() {
    call Leds.led2Toggle();
}
```

Para el correcto funcionamiento del programa deberemos iniciar el handler (manejador), y también iniciar las componentes que manejaremos. Asimismo

deberemos señalar que se han arrancado correctamente. Los búferes de almacenamiento para el envío de mensajes se inician y sus variables se ponen a cero.

```
event void Boot.booted() {
    uint8_t i;

    for (i = 0; i < UART_QUEUE_LEN; i++)
        uartQueue[i] = &uartQueueBufs[i];
    uartIn = uartOut = 0;
    uartBusy = FALSE;
    uartFull = TRUE;

    for (i = 0; i < RADIO_QUEUE_LEN; i++)
        radioQueue[i] = &radioQueueBufs[i];
    radioIn = radioOut = 0;
    radioBusy = FALSE;
    radioFull = TRUE;

    call RadioControl.start();
    call SerialControl.start();
}

event void RadioControl.startDone(error_t error) {
    if (error == SUCCESS) {
        radioFull = FALSE;
    }
}

event void SerialControl.startDone(error_t error) {
    if (error == SUCCESS) {
        uartFull = FALSE;
    }
}

event void SerialControl.stopDone(error_t error) {}
event void RadioControl.stopDone(error_t error) {}

uint8_t count = 0;
```

Cuando se produce alguno de estos eventos de recepción llegan mensajes vía radio del otro nodo dentro de la red. Se procede a almacenarlos dentro del búfer de mensajes que serán transmitidos por el puerto serie (UART) al PC.

```
message_t* ONE receive(message_t* ONE msg, void* payload,
uint8_t len);

event message_t *RadioSnoop.receive[am_id_t id](message_t
*msg,
void *payload,
uint8_t len) {
    return receive(msg, payload, len);
}
```

```

    event message_t *RadioReceive.receive[am_id_t id](message_t
*msg, void *payload,uint8_t len) {
    return receive(msg, payload, len);
    }

message_t* receive(message_t *msg, void *payload, uint8_t len)
{
    message_t *ret = msg;

    atomic {
        if (!uartFull)
        {
            ret = uartQueue[uartIn];
            uartQueue[uartIn] = msg;

            uartIn = (uartIn + 1) % UART_QUEUE_LEN;

            if (uartIn == uartOut)
                uartFull = TRUE;

            if (!uartBusy)
            {
                post uartSendTask();
                uartBusy = TRUE;
            }
        }
        else
            dropBlink();
    }

    return ret;
}

```

La tarea de envío a UART (Puerto Serie) es invocada para enviar los datos al PC a través del puerto serie. ENVIO DE DATOS PC-PUERTO SERIE

```

uint8_t tmpLen;

task void uartSendTask() {
    uint8_t len;
    am_id_t id;
    am_addr_t addr, src;
    message_t* msg;
    atomic
        if (uartIn == uartOut && !uartFull)
        {
            uartBusy = FALSE;
            return;
        }

    msg = uartQueue[uartOut];
    tmpLen = len = call RadioPacket.payloadLength(msg);
    id = call RadioAMPacket.type(msg);
    addr = call RadioAMPacket.destination(msg);
    src = call RadioAMPacket.source(msg);
    call UartPacket.clear(msg);
}

```

```

    call UartAMPacket.setSource(msg, src);

    if (call UartSend.send[id](addr, uartQueue[uartOut], len) ==
SUCCESS)
        call Leds.led1Toggle();
    else
    {
failBlink();
post uartSendTask();
    }
}

```

Se recoge el evento de envío correcto de mensaje a través de puerto serie y la liberación del búfer de envío UART, asimismo se vuelve a invocar al envío de datos por puerto serie.

```

event void UartSend.sendDone[am_id_t id](message_t* msg,
error_t error) {
    if (error != SUCCESS)
        failBlink();
    else
        atomic
        if (msg == uartQueue[uartOut])
        {
            if (++uartOut >= UART_QUEUE_LEN)
                uartOut = 0;
            if (uartFull)
                uartFull = FALSE;
        }
    post uartSendTask();
}

```

Este evento recoge los datos recibidos desde el PC a través de UART (puerto serie), y se almacenará el mensaje en el búfer de envío a través de radio al otro nodo de la red.

```

event message_t *UartReceive.receive[am_id_t id](message_t
*msg, void *payload, uint8_t len) {
    message_t *ret = msg;
    bool reflectToken = FALSE;

    atomic
    if (!radioFull)
    {
        reflectToken = TRUE;
        ret = radioQueue[radioIn];
        radioQueue[radioIn] = msg;
        if (++radioIn >= RADIO_QUEUE_LEN)
            radioIn = 0;
        if (radioIn == radioOut)
            radioFull = TRUE;

        if (!radioBusy)
        {
            post radioSendTask();
            radioBusy = TRUE;
        }
    }
}

```

```

    }
    else
        dropBlink();

    if (reflectToken) {
        //call UartTokenReceive.ReflectToken(Token);
    }

    return ret;
}

```

Llamada a la tarea de envío a través de radio. En ella se encapsulan dentro del mensaje el identificador, origen, dirección de destino y se envía el mensaje a través de radio al otro nodo de la red. Se hace nuevamente llamada a envío de radio.

```

task void radioSendTask() {
    uint8_t len;
    am_id_t id;
    am_addr_t addr, source;
    message_t* msg;

    atomic
    if (radioIn == radioOut && !radioFull)
    {
        radioBusy = FALSE;
        return;
    }

    msg = radioQueue[radioOut];
    len = call UartPacket.payloadLength(msg);
    addr = call UartAMPacket.destination(msg);
    source = call UartAMPacket.source(msg);
    id = call UartAMPacket.type(msg);

    call RadioPacket.clear(msg);
    call RadioAMPacket.setSource(msg, source);

    if (call RadioSend.send[id](addr, msg, len) == SUCCESS)
        call Leds.led0Toggle();
    else
    {
        failBlink();
        post radioSendTask();
    }
}

```

Se recoge el evento de envío correcto a través de la interfaz radio al otro nodo y se actualiza el búfer de envío a través de radio y se libera en caso de que estuviera lleno.

```

event void RadioSend.sendDone[am_id_t id](message_t* msg,
error_t error) {
    if (error != SUCCESS)
        failBlink();
    else
        atomic
        if (msg == radioQueue[radioOut])
        {

```

```

        if (++radioOut >= RADIO_QUEUE_LEN)
            radioOut = 0;
        if (radioFull)
            radioFull = FALSE;
    }

    post radioSendTask();
}
}

```

Una vez definido el sistema, se define un fichero de configuración (BaseStationC.nc) donde se indican todas las relaciones de componentes e interfaces en la implementación realizada:

```

configuration BaseStationC {
}
implementation {
    components MainC, BaseStationP, LedsC;
    components ActiveMessageC as Radio, SerialActiveMessageC as
Serial;

    MainC.Boot <- BaseStationP;

    BaseStationP.RadioControl -> Radio;
    BaseStationP.SerialControl -> Serial;

    BaseStationP.UartSend -> Serial;
    BaseStationP.UartReceive -> Serial;
    BaseStationP.UartPacket -> Serial;
    BaseStationP.UartAMPacket -> Serial;

    BaseStationP.RadioSend -> Radio;
    BaseStationP.RadioReceive -> Radio.Receive;
    BaseStationP.RadioSnoop -> Radio.Snoop;
    BaseStationP.RadioPacket -> Radio;
    BaseStationP.RadioAMPacket -> Radio;

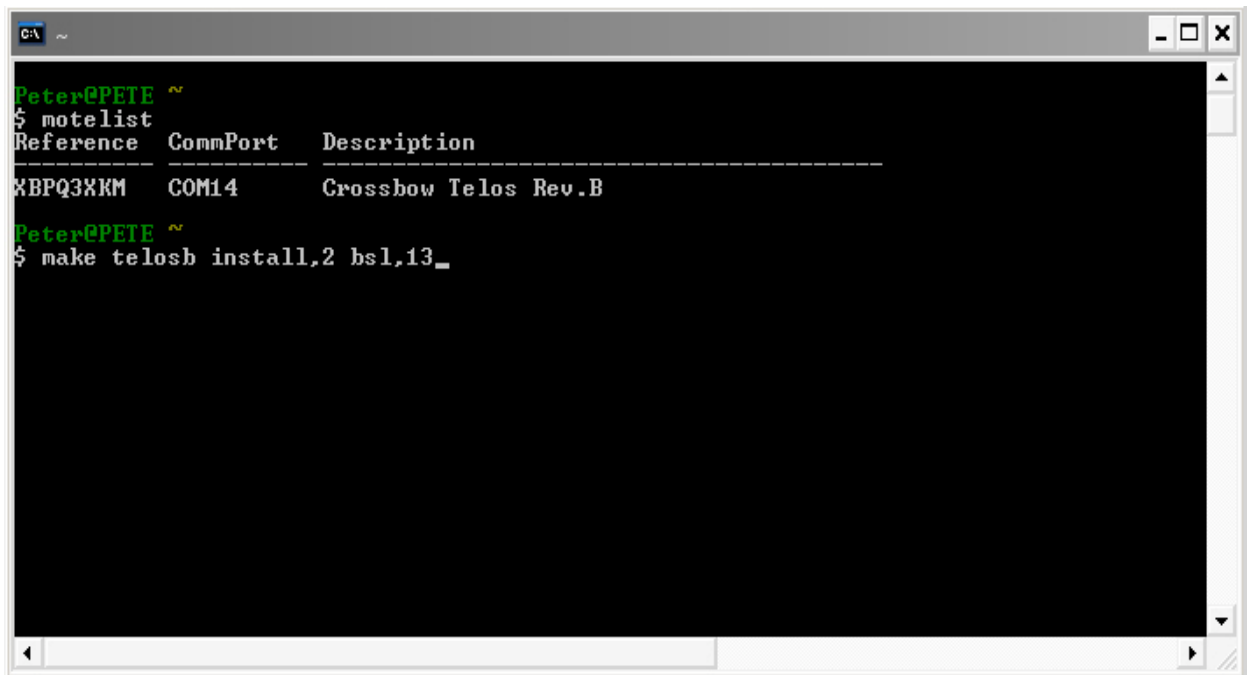
    BaseStationP.Leds -> LedsC;
}

```

Una vez hecho todo esto, se compila el código y se carga en el dispositivo TelosB empleando los siguientes comandos en la consola de Cygwin:

> *motelist* (con ésto podremos extraer el puerto serie al que está conectado el dispositivo)

> *make telosb install,2 bsl,NºPuerto Serie - 1*



```
Peter@PETE ~
$ motelist
Reference  CommPort  Description
-----
XBPQ3XKM  COM14     Crossbow Telos Rev.B
Peter@PETE ~
$ make telosb install,2 bs1,13_
```

Figura 19 – Consola de Cygwin compilación BaseStation

El dispositivo queda programado para ser conectado al PC, el cual a través del puerto USB le provee la energía necesaria para funcionar.

1.2 Oscilloscope

Oscilloscope será el programa encargado de tomar un número determinado de muestras de temperatura del sensor de temperatura montado en el dispositivo TelosB con un tiempo de muestreo determinado por el usuario, e ir enviándoselas al otro nodo de la red cada vez que se cumpla dicho intervalo de tiempo.

También es capaz el programa de leer mensajes enviados desde el otro nodo para actualizar su versión, o modificar las opciones de la toma de muestras de temperatura.

Trama de envío:

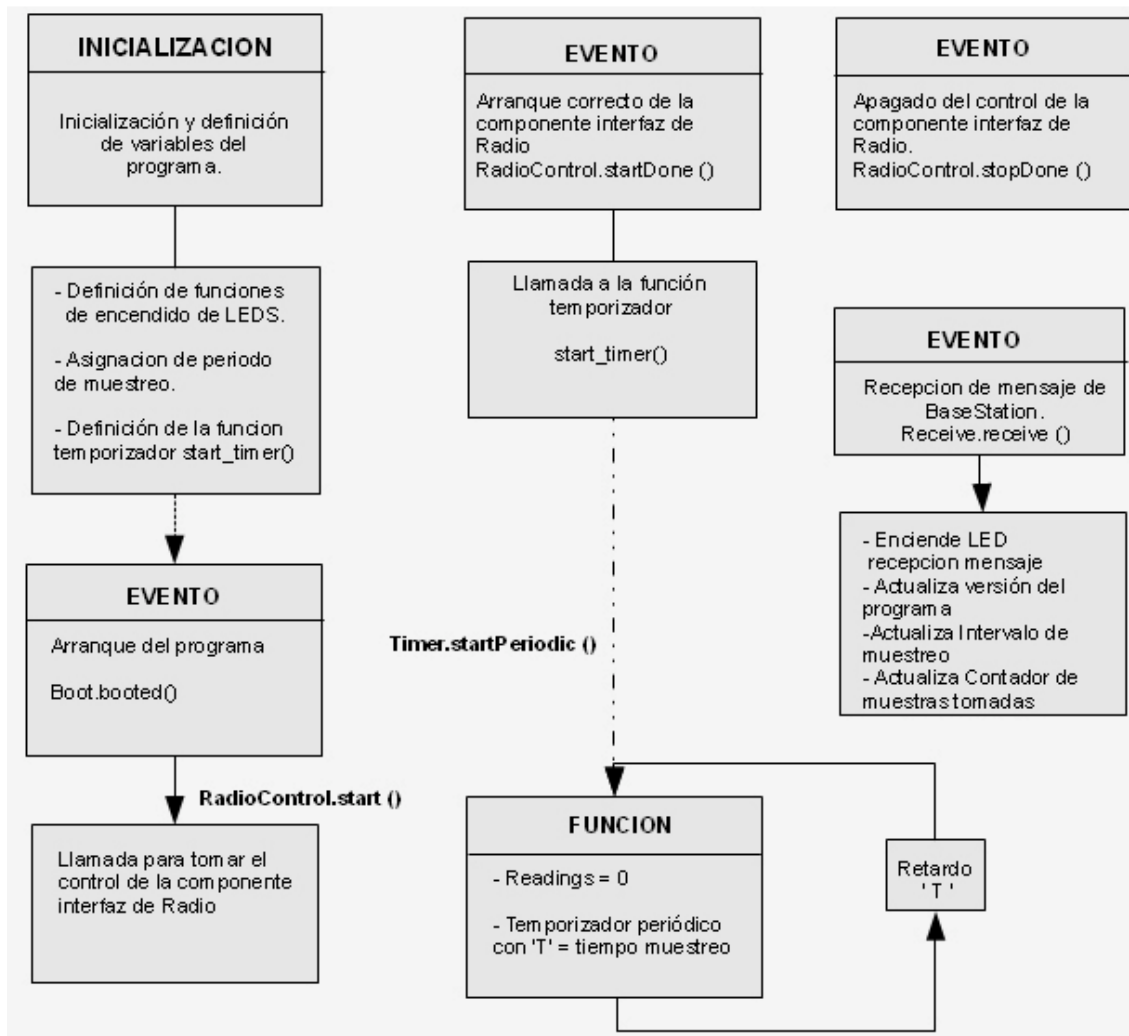
El mensaje enviado desde Osciloscopio al otro nodo BaseStation corresponde a una estructura del tipo *message_t*, cuyo tamaño del campo de datos está configurado por defecto en 28 bytes.

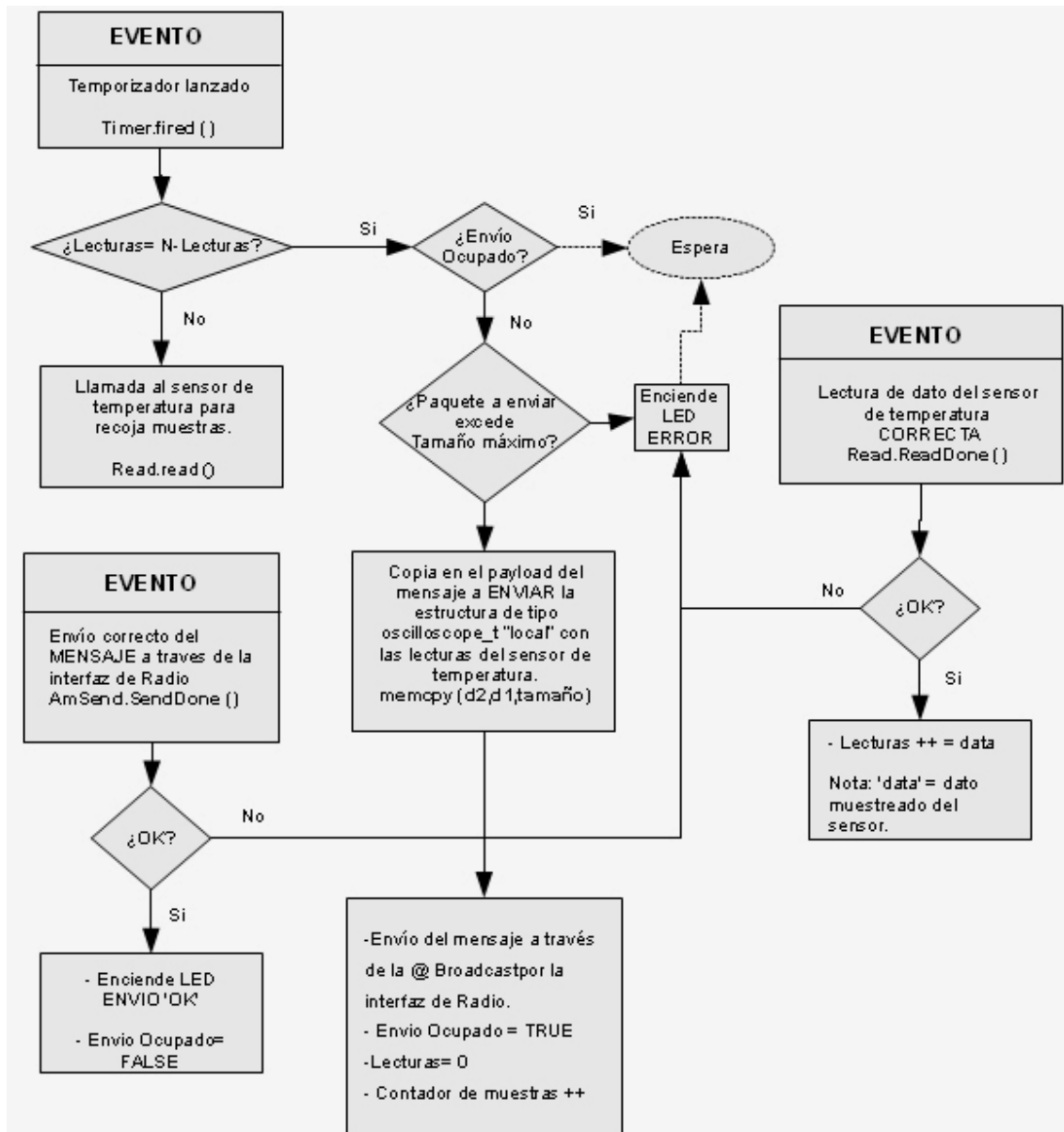
Para guardar las muestras recogidas del sensor de temperatura dispone Osciloscopio de una segunda estructura, del tipo *oscilloscope_t*. La aplicación irá guardando las muestras en una estructura *oscilloscope_t* para posteriormente encapsularlas en una estructura del tipo *message_t* y ser enviadas vía Radio.

La estructura *oscilloscope_t* tendrá los siguientes campos, los cuales son enteros sin signo de 16 bits:

- **Versión:** versión del programa Osciloscopio.
- **Interval: Periodo de sampleo del sensor.** Éste valor es importante porque determina cada cuánto tiempo se envían las tramas con las muestras recogidas al otro dispositivo. Éste valor es configurable.
- **Id:** Identificador del dispositivo que envía las muestras al dispositivo receptor. Nos sirve para crear una red con varios nodos sensores enviando a un nodo receptor.
- **Count:** contador de muestras recogidas.
- **Lecturas [Nº Lecturas] :** Array de datos con las lecturas recogidas del sensor de temperatura. El tamaño del array es configurable. Por defecto el dispositivo recogerá 10 muestras por cada intervalo de tiempo, este valor es configurable.

Diagrama de bloques de Osciloscopio:





Implementación:

Los ficheros correspondientes a esta implementación se encuentran en el directorio \apps\Oscilloscope del directorio raíz.

La implementación se desarrolla en el fichero OscilloscopeC.nc.

Primeramente cargamos las interfaces necesarias para el control del dispositivo TelosB

```
/**
 * Osciloscopio Sensor Temperatura
```

```

*/
#include "Timer.h"
#include "Oscilloscope.h"
module OscilloscopeC
{
uses {

interface Boot;
interface SplitControl as RadioControl;
interface AMSend;
interface Receive;
interface Timer<TMilli>;
interface Read<uint16_t>;
interface Leds;

}
}

```

A continuación se definen las variables que vamos a utilizar en nuestro programa.

```

implementation
{
message_t sendBuf;
bool sendBusy;

/* Estado actual local - intervalo, version y lecturas
acumuladas */
oscilloscope_t local;

uint8_t reading; /* De 0 a NREADINGS */

/* Cuando recibimos un mensaje en Osciloscopio del otro
extremo comprobamos que su contador de muestras sea igual. Si es
superior al nuestro, coloca nuestro contador igualado al
contador recibido en el mensaje, entonces debemos suprimir
nuestro próximo incremento del contador propio. Ésta es una
forma muy simple de sincronización. */

bool suppressCountChange;

```

En este momento se crean funciones de llamada al control de los leds del dispositivo para reportar el envío o recepción de mensajes y si ha habido algún problema.

```

// Usar LEDs para reportar distintos tipos de estado.
void report_problem() { call Leds.led0Toggle(); }
void report_sent() { call Leds.led1Toggle(); }
void report_received() { call Leds.led2Toggle(); }

```

A continuación se arranca el programa y se inician también los componentes que van a ser usados. Asimismo se verifica con un evento que se ha arrancado con éxito, lo cual empieza a lanzar el temporizador con el tiempo de muestreo determinado.

```

event void Boot.booted() {
local.interval = DEFAULT_INTERVAL;

```

```

    local.id = TOS_NODE_ID;
    if (call RadioControl.start() != SUCCESS)
        report_problem();
}

void startTimer() {
    call Timer.startPeriodic(local.interval);
    reading = 0;
}

event void RadioControl.startDone(error_t error) {
    startTimer();
}

event void RadioControl.stopDone(error_t error) {
}

```

Este evento se produce cuando se recibe un mensaje del otro nodo de la red. En él se recibe la versión del programa, el intervalo de tiempo de muestreo deseado, y el contador para sincronizar la aplicación.

```

event message_t* Receive.receive(message_t* msg, void*
payload, uint8_t len) {
    oscilloscope_t *ormsg = payload;

    report_received();

    /* Si recibimos una nueva version actualize nuestro
    interval.
    Si observamos un contador futuro, salta adelante y
    suprime nuestro cambio propio
    */
    if (ormsg->version > local.version)
    {
        local.version = ormsg->version;
        local.interval = ormsg->interval;
        startTimer();
    }
    if (ormsg->count > local.count)
    {
        local.count = ormsg->count;
        suppressCountChange = TRUE;
    }

    return msg;
}

```

A continuación se procede a la lectura y copia de muestras en un mensaje para ser enviado.

```

/* - Si el buffer local de muestreo esta completo, envía las
muestras acumuladas.
- Leer la siguiente muestra.
*/
event void Timer.fired() {
    if (reading == NREADINGS)
    {

```

```

        if (!sendBusy && sizeof local <= call
AMSend.maxPayloadLength())
        {
            // No es necesario comprobar que sea nulo porque hemos
            // chequeado ya el tamaño más arriba.

            memcpy(call AMSend.getPayload(&sendBuf, sizeof(local)),
&local, sizeof local);
            if (call AMSend.send(AM_BROADCAST_ADDR, &sendBuf,
sizeof local) == SUCCESS)
                sendBusy = TRUE;
        }
        if (!sendBusy)
            report_problem();

        reading = 0;

        /* Parte 2 de "sincronizacion de tiempo sencilla":
incrementar nuestro contador si no lo hicimos antes. */
        if (!suppressCountChange)
            local.count++;
        suppressCountChange = FALSE;
    }

    // Llamada a lectura de datos del sensor.

    if (call Read.read() != SUCCESS)
        report_problem();
}

```

A continuación se recoge un evento de envío correcto de los datos al otro nodo dentro de nuestra red y se libera el canal de radio.

```

event void AMSend.sendDone(message_t* msg, error_t error) {
    if (error == SUCCESS)
        report_sent();
    else
        report_problem();

    sendBusy = FALSE;
}

```

Finalmente se recoge un evento de lectura correcta del sensor de temperatura, y si es correcto, se procede a rellenar el búfer de muestras con nuevas lecturas del sensor.

```

event void Read.readDone(error_t result, uint16_t data) {
    if (result != SUCCESS)
    {
        data = 0xffff;
        report_problem();
    }
}

```

```

    }
    local.readings[reading++] = data;
  }
}

```

A continuación se crea un fichero de encabezado con los valores de la aplicación (**Oscilloscope.h**) donde se indican el número de lecturas por mensaje enviado, el intervalo por defecto de envío de muestras y el identificador de Active Message de la aplicación.

Dentro del archivo *Oscilloscope.h* se define la estructura de soporte principal del programa. Se define el tipo *oscilloscope_t*. Contendrá dentro de ella varios valores enteros sin signo de 16 bits: la versión, el intervalo de tiempo entre envíos de muestras (por defecto 256 milisegundos), identificador del dispositivo que envía las muestras, un contador de muestras leídas y un array donde se irán guardando las lecturas realizadas (por defecto se recoge un array de 10 muestras).

Esta estructura *oscilloscope_t* es encapsulada dentro de una estructura *message_t* en su campo de Datos para ser enviada al otro nodo BaseStation.

```

#ifndef OSCILLOSCOPE_H
#define OSCILLOSCOPE_H

enum {
  /* Number of readings per message. If you increase this, you
  may have to
  increase the message_t size. */
  NREADINGS = 10,

  /* Default sampling period. */
  DEFAULT_INTERVAL = 256,

  AM_OSCILLOSCOPE = 0x93
};

typedef nx_struct oscilloscope {
  nx_uint16_t version; /* Version of the interval. */
  nx_uint16_t interval; /* Sampling period. */
  nx_uint16_t id; /* Mote id of sending mote. */
  nx_uint16_t count; /* The readings are samples count *
  NREADINGS onwards */
  nx_uint16_t readings[NREADINGS];
} oscilloscope_t;

#endif

```

Una vez definido el sistema, se crea un fichero de configuración (*OscilloscopeAppC.nc*) donde se indican todas las relaciones de componentes e interfaces en la implementación realizada. Podemos comprobar cómo las lecturas (llamada a “Read”) que realiza nuestra aplicación son relación con la componente del sensor de temperatura *SensirionSht11* y su interfaz de lectura:

```

configuration OscilloscopeAppC { }
implementation
{
    components OscilloscopeC, MainC, ActiveMessageC, LedsC,
        new TimerMilliC(), new SensirionSht11C(),
        new AMSenderC(AM_OSCILLOSCOPE), new
AMReceiverC(AM_OSCILLOSCOPE);

    OscilloscopeC.Boot -> MainC;
    OscilloscopeC.RadioControl -> ActiveMessageC;
    OscilloscopeC.AMSend -> AMSenderC;
    OscilloscopeC.Receive -> AMReceiverC;
    OscilloscopeC.Timer -> TimerMilliC;
    OscilloscopeC.Read -> SensirionSht11C.Temperature;
    OscilloscopeC.Leds -> LedsC;
}

```

Una vez hecho todo esto, se compilará el código y se carga en el dispositivo TelosB empleando los siguientes comandos en la consola de Cygwin:

> *motelist* (con ésto podremos extraer el puerto serie al que está conectado el dispositivo)

> *make telosb install,1 bsl,NºPuerto Serie - 1*

```

/opt/tinyos-2.x/apps/OscilloscopePedro
XB020XTU COM6 Crossbow Telos Rev.B
Peter@PETE /opt/tinyos-2.x/apps/OscilloscopePedro
$ make telosb install,1 bsl,5
mkdir -p build/telosb
compiling OscilloscopeAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmul -Wall -Wshadow -Wnesc-all -target=telosb -fne
TOS_AM_GROUP=0x22 -DIDENT_APPNAME="OscilloscopeApp" -DIDENT_USERNAME="Peter" -DIDENT_HOSTNAME
T_TIMESTAMP=0x4b4e044dL -DIDENT_UIDHASH=0xe40be1b7L OscilloscopeAppC.nc -lm
/opt/tinyos-2.x/tos/chips/cc2420/lpl/DummyLplC.nc:39:2: warning: #warning "*** LOW POWER COMMUNICA
compiled OscilloscopeAppC to build/telosb/main.exe
13426 bytes in ROM
394 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
writing IOS image
telos-set-symbols --objcopy msp430-objcopy --objdump msp430-objdump --target ihex build/telosb/main.
=1 ActiveMessageAddressC$addr=1
installing telosb binary using bsl
telos-bsp --telosb -c 5 -r -e -I -p build/telosb/main.ihex.out-1
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
13458 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-1 build/telosb/main.ihex.out-1
$

```

Figura 20 – Consola de Cygwin compilación Oscilloscope

El dispositivo TelosB quedará ya programado y listo para ser activado mediante la energía de las baterías que le proveamos.

Una vez hecho esto el nodo con el programa Osciloscopio tomará muestras y empezará a enviar al nodo BaseStation que a su vez las pasará a través del puerto serie al PC.

Para poder visualizar dichas muestras se han implementado dos programas en Java que nos permitirán controlar el sensor de temperatura, su monitorización y el envío a través del puerto serie de forma correcta de los paquetes al PC.

1.3. SerialForwarder

Uno de los problemas de usar directamente el puerto serie del PC es que solo un programa en el PC puede interactuar con el dispositivo conectado. Además requiere que carguemos la aplicación en el PC que está conectado físicamente al dispositivo. La herramienta SerialForwarder permite saltarse estas limitaciones.

SerialForwarder es una aplicación Java que nos permite unir puertos de comunicaciones como el "Com", "Tossim-radio", "Tossim-Uart" con un puerto TCP de un PC, de manera que sirve de pasarela entre ambos. Es muy útil ya que permite realizar una aplicación PC que envíe o reciba datos por TCP/IP, y estos datos ser recibidos por SerialForwarder y reenviados al puerto de comunicaciones que le hayamos especificado y viceversa, recibir datos de un dispositivo sensor conectado por el puerto COM, y ser reenviados via TCP/IP con SerialForwarder.

Para nuestra aplicación de medición de temperatura para poder compilarlo y arrancarlo accederemos a la carpeta Java dentro de TinyOS, que se encuentra en:

```
/tinyos-2.x/support/sdk/java/
```

Y escribiremos por consola:

```
>make ( si no hemos compilado Java, deberíamos hacerlo al instalarlo en  
Cygwin (Windows) o en Xubuntos (Linux))  
>motelist (para saber de nuevo el puerto al que está conectado nuestro  
mote  
estación base)  
>java net.tinyos.sf.SerialForwarder -comm  
serial@/dev/ttyUSBx:telosb &
```

Con ello se lanzará la interfaz gráfica del programa, donde observaremos que empezaremos a recibir paquetes de nuestro otro dispositivo dentro de la red, el otro dispositivo TelosB, que carga el programa Osciloscopio.

En la figura podemos ver la interfaz del programa:

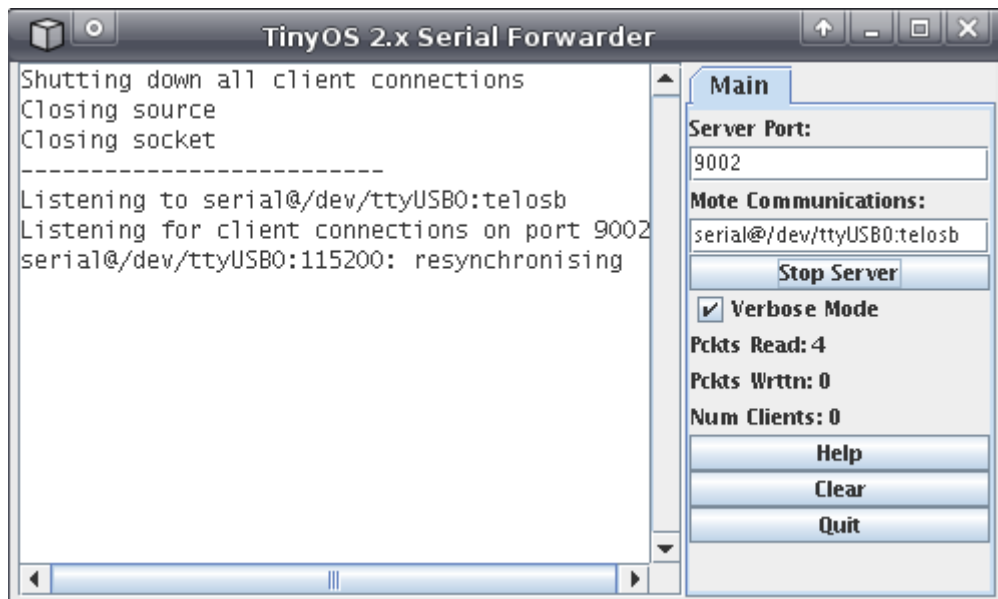


Figura 21 - GUI (Graphic User Interface) de Serial Forwarder

En la imagen podemos apreciar los paquetes recibidos, los enviados en el caso de que enviemos algún cambio de configuración al otro dispositivo dentro de la red, y la configuración del puerto al que se conecta y el puerto TCP por el que emite.

1.4. Herramienta Java de Osciloscopio

Disponemos de una aplicación Java dentro de la carpeta del Osciloscopio para extraer por pantalla los datos que nos llegan del sensor de temperatura SHT11 del dispositivo TelosB. Ésta aplicación se arranca desde el PC, y por lo tanto no tiene que ver con la aplicación que corre en el nodo inalámbrico que recoge muestras. Recoge los datos recibidos por el puerto serie de BaseStation y los muestra por pantalla.

La aplicación se encuentra en:

```
\apps\Oscilloscope\java\
```

Para lanzar la aplicación deberemos escribir en la consola de Cygwin (o en la consola de Xubuntu en Linux):

```
> cd apps
> cd Oscilloscope
> cd java
> ./run
```

La aplicación nos provee una interfaz gráfica de usuario que nos permitirá monitorizar las lecturas de temperatura, modificar los rangos de la gráfica donde se muestra la temperatura, y alterar la velocidad de muestreo del dispositivo sensor.

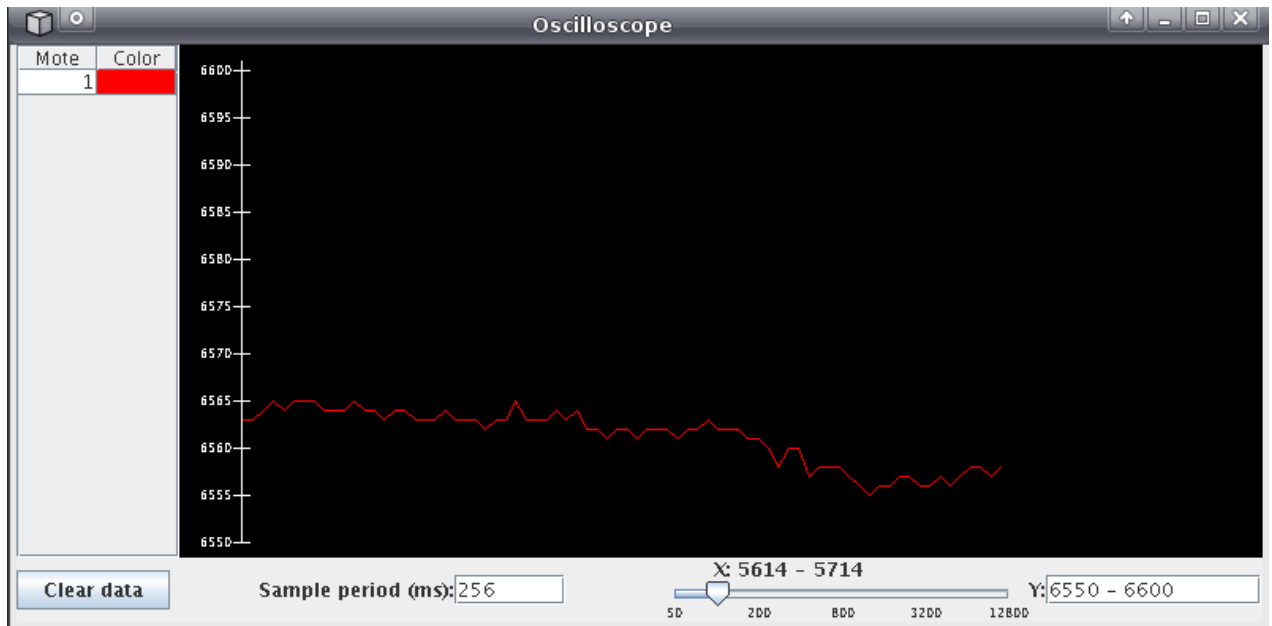


Figura 22 - Vista principal de la GUI de la aplicación Java de Osciloscopio

Los controles en la inferior de la pantalla nos permiten hacer Zoom en el eje X, cambiar el rango del eje Y o borrar todos los datos recibidos. Podemos modificar el color usado para presentar las muestras del dispositivo clickeando en su color en la tabla de dispositivos conectados.

Para calcular la temperatura en grados centígrados debemos recordar las especificaciones dadas en el apartado 3.2 (Sensirion SHT-11) del Capítulo 2 (Estado del Arte).

2. Problemática de TinyOS 2.0.2

Desde la versión 1.1.3 de TinyOS utiliza una capa MAC llamada B-MAC. Este algoritmo MAC está implementado para las comunicaciones del chip de radio CC2420, que es el que incorporan los dispositivos TelosB desde la versión 1.1.7 de TinyOS. Sin embargo aunque la radio CC2420 no tiene implementada la funcionalidad de escucha de baja potencia (Low Power Listening), si existen las interfaces MacControl y MacBackoff, pero no están implementadas.

B-MAC proporciona una interfaz para obtener menor consumo de energía, evita colisiones y hace una buena utilización, aunque no óptima, del canal de comunicaciones.

Para conseguir operar con baja potencia, B-MAC emplea un esquema adaptativo de muestreo de preámbulo para reducir el ciclo de ocupación y minimizar la escucha en vano. B-MAC soporta reconfiguración sobre la marcha y provee interfaces bidireccionales para los servicios del sistema para optimizar su actuación, ya sea por rendimiento, latencia, o conservación de energía. B-MAC respecto a 802.11 convencional se muestra con mejores tasas de envío, rendimiento, latencia y consumo de energía que S-MAC.

Pero B-MAC es 802.11. Así que no está enfocado al completo como lo está 802.15.4 a las redes de sensores inalámbricos de baja potencia (LR-WPAN), sino que es una adaptación.

Y asimismo dentro de TinyOS tanto la capa MAC como la capa física (PHY) están precompiladas, por lo que no podremos modificar nada de ellas para intentar sacar un mayor rendimiento en nuestra red: tasa de bit, tamaño de paquete, etc.

Todo ello sí es posible utilizando Open Zigbee, que actualmente es la única implementación real de 802.15.4 dentro de TinyOS y hace uso de las características del protocolo específico para redes de sensores inalámbricos (WSN).

Por ello vamos a desarrollar nuestra aplicación sensor de temperatura dentro del entorno de programación Open Zigbee.

3. Desarrollo con Open Zigbee 2.0 Beta

Esta es la solución final utilizada en éste Proyecto Fin de Carrera.

Para trabajar con Open Zigbee deberemos de instalar nuestra implementación de Open Zigbee 2.0 que está en una fase beta, ya que para trabajar sobre TinyOS 2.0.2 deberemos usar dicha versión.

Deberemos copiar la carpeta hurray2x de la pagina web de Open Zigbee y meterla dentro del directorio de software contributivo de TinyOS 2.x:

```
\tinynos-2.x-contrib\
```

Una vez hecho esto, ya tenemos instalado Open Zigbee. A continuación se procede a comprobar las dependencias de componentes que tienen en TinyOS nuestras aplicaciones. Comprobamos como podrían surgirnos problemas a la hora de que no se hayan implementado dichas componentes en Open Zigbee.

3.1 Transmisión de mensajes por medio de Active Messages

Tanto nuestra aplicación BaseStation, que se encargaba de enviar los mensajes recibidos a través del puerto serie al PC, como nuestra aplicación Oscilloscope, que se encarga de extraer las muestras del sensor de temperatura, hacen uso del paradigma de los *ActiveMessage* para las transmisiones de mensajes entre ellos.

Ello consiste en que los mensajes (una estructura del tipo `message_t`) que se transmiten los distintos dispositivos poseen cada uno un campo de *payload* (los datos en claro) y un ID de manejador (handler ID), que es invocado al recibir un mensaje del dispositivo que está en la red inalámbrica. Dicho ID será un entero. Es un número de puerto que va en la cabecera del mensaje. Cuando llega un mensaje a un dispositivo, el evento asociado con ese ID es lanzado. Diferentes dispositivos pueden asociar diferentes eventos de recepción con el mismo identificador.

Las transmisiones de mensajes las realizan los dispositivos TelosB a través del chip de radio CC2420.

La componente `ActiveMessageC.nc`, que es la encargada de dichas llamadas de Radio en TinyOS, es dependiente del Hardware, es decir, para la plataforma TelosB disponemos de un `ActiveMessageC.nc`, que será distinto al de otras plataformas.

Dicho componente lo podemos encontrar en:

```
\tos\platform\telosa
```

Las dependencias entre componentes para la transmisión de mensajes via radio sería tal como la figura:

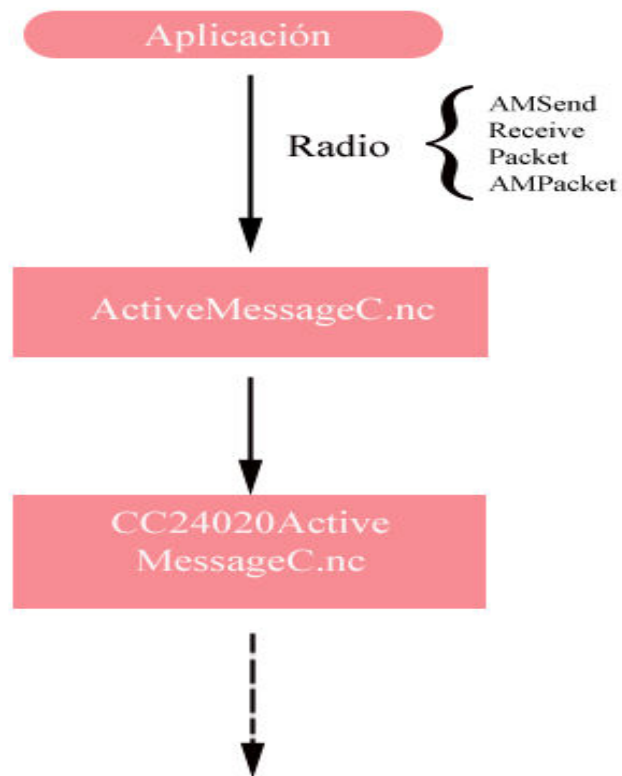


Figura 23 - Dependencias de componentes para el uso del chip de radio CC2420 en TinyOS

Para manejar el chip de radio en Open Zigbee encontramos la dificultad de que no está implementado el soporte para el chip de radio con Active Messages, ya que si bien está implementada la componente ActiveMessageC para la plataforma TelosB, no lo está el componente de la que dependen las interfaces comunes de radio (AMSend, Receive, Packet o AMPacket, ...) que se llama CC2420ActiveMessageC.nc, y se encuentra en TinyOS.

En cambio sí esta implementado en Open Zigbee la transmisión de Active Messages a través de la transmisión por puerto serie.

Por esto para poder transmitir los mensajes dentro de la red se ha optado por hacer uso de las interfaces y componentes que nos proveen Open Zigbee.

3.2. Interfaz de envío de datos en OpenZigbee: MCPS_DATA

En Open Zigbee disponemos de una interfaz llamada MCPS_DATA.nc, que está implementada en la componente Mac.nc y MacM.nc.

Dentro de MCPS_DATA disponemos de:

- MCPS_DATA.request – Comando de envío. Este comando es el que nos permite enviar datos a otro dispositivo en una red inalámbrica. Debemos rellenar los

siguientes campos: Dirección de Origen, Dirección de destino. Identificador de la red de destino (PANID), el mensaje en sí, que será un array de enteros de 8 bits, y las opciones de transmisión.

- MCPS_DATA.confirm – Evento de confirmación. Este evento se lanza para recoger el estado del envío de un mensaje, en caso de ser requerido.
- MCPS_DATA.indication – Evento de recepción. Este evento se lanza cuando llegan datos de otro dispositivo. Los datos en claro llegan en un array de enteros sin signo de 8 bit.

Ahora pasamos a ver la modificación de las aplicaciones que creamos para TinyOS para poder transmitir los datos del sensor de temperatura al otro nodo dispositivo en la red.

3.3 Osciloscopio en Open-Zigbee

Como recordamos esta herramienta se encarga de extraer muestras de un sensor de temperatura-humedad SHT11, las encapsula en una estructura que es enviada al otro dispositivo en nuestra red.

El intervalo de tiempo entre el envío de muestras está establecido por defecto en 128 milisegundos. Y las muestras enviadas serán por defecto 10. Estos valores son configurables en el archivo *Oscilloscope.h*.

Hemos modificado la aplicación Osciloscopio con el fin de que haga uso de las interfaces de envío y recepción via Radio que provee Open Zigbee.

Implementación:

Los ficheros correspondientes a ésta implementación se encuentran en la carpeta `\apps\OscilloscopeHurray\` del directorio raíz.

Primeramente detallamos la implementación realizada. El fichero con la implementación se llama *OscilloscopeAppC.nc*.

Lo primero que se hace es cargar las interfaces que van a ser usadas por nuestro dispositivo TelosB, entre ellas las interfaces MAC de Open Zigbee.

```
#include <Timer.h>
#include "Oscilloscope.h"

module OscilloscopeC
{
    uses interface Boot;
    uses interface SplitControl as RadioControl;
    uses interface Timer<TMilli> as TimerPrinc;
    uses interface Timer<TMilli> as Timer0;

    uses interface Read<uint16_t>;
```

```

// MAC interfaces
  uses interface Leds;
  uses interface MLME_START;
  uses interface MLME_GET;
  uses interface MLME_SET;
  uses interface MLME_BEACON_NOTIFY;
  uses interface MLME_GTS;
  uses interface MLME_ASSOCIATE;
  uses interface MLME_DISASSOCIATE;
  uses interface MLME_ORPHAN;
  uses interface MLME_SYNC;
  uses interface MLME_SYNC_LOSS;
  uses interface MLME_RESET;
  uses interface MLME_SCAN;
  uses interface MCPS_DATA;
}

```

A continuación se definen las variables que van a ser usadas globalmente por nuestro programa: Dirección de envío, de recepción, array de datos a enviar, dirección corta, lecturas del sensor.

```

implementation {

    PANDescriptor pan_des;

    uint32_t my_short_address=0x00000000;

    uint32_t DestinationMote[2];
    uint32_t SourceMoteAddr[2];
    uint8_t v_temp[2];

    uint8_t msdu_payload[28];

    bool sendBusy;
    oscilloscope_t local;

    uint8_t reading; /* 0 to NREADINGS */

    /* Cuando recibimos un mensaje en Osciloscopio del otro extremo
    comprobamos que su contador de muestras sea igual. Si es
    superior al nuestro, coloca nuestro contador igualado al
    contador recibido en el mensaje, entonces debemos suprimir
    nuestro próximo incremento del contador propio. Ésta es una
    forma muy simple de sincronización.
    */

    bool suppressCountChange = FALSE;
}

```

Aviso mediante llamadas a función para el encendido de los leds en los distintos estados del dispositivo.

```
// Usa LEDs para reportar varios tipos de estado
void report_problem() {call Leds.led0Toggle();}
void report_sent() { call Leds.led1Toggle(); }
void report_received() { call Leds.led2Toggle(); }
```

Arranque del dispositivo, se definen las variables de dirección de destino y se vacía el array de muestras. Se arranca la componente radio del dispositivo y se lanza un temporizador

```
event void Boot.booted() {

    DestinationMote[0]=0x00000000;
    DestinationMote[1]=0x00000000;
    msdu_payload[0]=0x00;
    my_short_address = 0x0000;
    local.interval=DEFAULT_INTERVAL;
    local.id = TOS_NODE_ID;

    if (call RadioControl.start() != SUCCESS)
        report_problem();

    call Timer0.startOneShot(4000);

}
```

El evento recogido cuando se acaba el temporizador anterior nos permite dar una dirección corta MAC a nuestro dispositivo y un PANID a nuestro dispositivo que serán variables.

```
event void Timer0.fired() {

    my_short_address = TOS_NODE_ID;

    //Establece la direccion corta MAC variable
    v_temp[0] = (uint8_t) (my_short_address >> 8);
    v_temp[1] = (uint8_t) (my_short_address );

    call MLME_SET.request (MACSHORTADDRESS,v_temp);

    //Establece el MAC PANID variable

    v_temp[0] = (uint8_t) (MAC_PANID >> 8);
    v_temp[1] = (uint8_t) (MAC_PANID );

    call MLME_SET.request (MACPANID,v_temp);

}
```

Si se ha arrancado bien la componente de radio, se llama a la función `startTimer`, que a su vez lanzará el temporizador con tiempo de muestreo por defecto para tomar muestras del sensor de temperatura, y pondrá las lecturas a cero.

```
// TIEMPO DE MUESTREO

void startTimer() {
    call TimerPrinc.startPeriodic(local.interval);
    reading = 0;
}

event void RadioControl.startDone(error_t error) {
    startTimer();
}

event void RadioControl.stopDone(error_t error) {
}
```

Cuando recibamos algún mensaje del otro dispositivo de la red, tomaremos la dirección de origen del mensaje dentro del evento, y la copiamos a la dirección de envío de mensajes para nuestro posterior envío de muestras.

```
event error_t MCPS_DATA.indication(uint16_t SrcAddrMode,
uint16_t SrcPANId, uint32_t SrcAddr[2], uint16_t DstAddrMode,
uint16_t DestPANId, uint32_t DstAddr[2], uint16_t
msduLength, uint8_t msdu[28], uint16_t mpduLinkQuality, uint16_t
SecurityUse, uint16_t ACLEntry)
{
    DestinationMote[0]=SrcAddr[0];
    DestinationMote[1]=SrcAddr[1];

    return SUCCESS;
}
```

A continuación cada vez que se lanza el temporizador con el periodo de muestreo configurado, por defecto 256 mseg, se copian en el array de envío de datos las muestras tomadas por el sensor para ser enviadas, por defecto 10 muestras por intervalo, y se procede a enviar las muestras al otro dispositivo de nuestra red. Se actualiza el contador de muestras y se vuelve a llamar para tomar muestras o lecturas del sensor.

```
event void TimerPrinc.fired() {

    if (reading == NREADINGS)
    {
        if (!sendBusy && sizeof local <= sizeof msdu_payload)
        {

            memcpy( &msdu_payload, &local, sizeof local);
```



```

// #####
// ENVIO DE BUFFER DE DATOS POR RADIO
// #####

// está cambiado para que tome por destino la dirección que
// le llega en MCPS_DATA.indication del otro dispositivo.

// DestinationMote[0]=0x00000000;
// DestinationMote[1]=0x00000000;

SourceMoteAddr[0]=0x00000000;
SourceMoteAddr[1]=TOS_NODE_ID;

call MCPS_DATA.request(SHORT_ADDRESS, MAC_PANID,
SourceMoteAddr, SHORT_ADDRESS, MAC_PANID, DestinationMote,
1, msdu_payload,1,set_txoptions(0,0,0,0));
report_sent();
}

reading = 0;

/* Parte 2 de sincronizador de tiempo: incrementa nuestro
contador si nosotros no habiamos
saltado adelante el contador*/

if (!suppressCountChange)
    local.count++;
suppressCountChange = FALSE;
}

// LLAMADA A LECTURA DE DATOS DEL SENSOR

if (call Read.read() != SUCCESS)
    report_problem();

}

```

Este evento recoge que se ha leído correctamente del sensor de temperatura-humedad. Pone las muestras dentro de una estructura del tipo *Oscilloscope_t*

```

// EVENTO DE CONFIRMACION DE LECTURA DE DATOS DEL SENSOR

event void Read.readDone(error_t result, uint16_t data) {
    if (result != SUCCESS)
    {
        data = 0xffff;
        report_problem();
    }
    local.readings[reading++] = data;
}

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

```

```

/*****MLME-
SCAN*****/
/*****/
event error_t MLME_SCAN.confirm(uint8_t status,uint8_t ScanType,
uint32_t UnscannedChannels, uint8_t ResultListSize, uint8_t
EnergyDetectList[], SCAN_PANDescriptor PANDescriptorList[])
{

    return SUCCESS;
}

/*****MLME-
ORPHAN*****/
/*****/
event error_t MLME_ORPHAN.indication(uint32_t OrphanAddress[1],
uint8_t SecurityUse, uint8_t ACLEntry)
{

    return SUCCESS;
}

/*****MLME-
RESET*****/
/*****/
event error_t MLME_RESET.confirm(uint8_t status)
{

    return SUCCESS;
}

/*****MLME-SYNC-
LOSS*****/
/*****/
event error_t MLME_SYNC_LOSS.indication(uint8_t LossReason)
{

    return SUCCESS;
}

/*****MLME-
GTS*****/
/*****/
event error_t MLME_GTS.confirm(uint8_t GTSCharacteristics,
uint8_t status)
{

    return SUCCESS;
}

```

```

}

event error_t MLME_GTS.indication(uint16_t DevAddress, uint8_t
GTSCharacteristics, bool SecurityUse, uint8_t ACLEntry)
{
    return SUCCESS;
}

/*****
*****/
/*****MLME-BEACON
NOTIFY*****/
/*****
*****/
event error_t MLME_BEACON_NOTIFY.indication(uint8_t
BSN,PANDescriptor pan_descriptor, uint8_t PenAddrSpec, uint8_t
AddrList, uint8_t sduLength, uint8_t sdu[])
{

    return SUCCESS;
}

/*****
*****/
/*****MLME-
START*****/
/*****
*****/
event error_t MLME_START.confirm(uint8_t status)
{

return SUCCESS;
}

/*****
*****/
/*****MLME-SET
*****/
/*****
*****/

event error_t MLME_SET.confirm(uint8_t status,uint8_t
PIBAttribute)
{

return SUCCESS;
}

/*****
*****/
/*****MLME-GET
*****/
/*****
*****/
event error_t MLME_GET.confirm(uint8_t status,uint8_t
PIBAttribute, uint8_t PIBAttributeValue[])
{

return SUCCESS;
}
}

```

```

/*****
*****/
/*****MLME-
ASSOCIATE*****/
/*****/
event error_t MLME_ASSOCIATE.indication(uint32_t
DeviceAddress[], uint8_t CapabilityInformation, bool
SecurityUse, uint8_t ACLEntry)
{

    return SUCCESS;
}

event error_t MLME_ASSOCIATE.confirm(uint16_t AssocShortAddress,
uint8_t status)
{

    return SUCCESS;
}

/*****
*****/
/*****MLME-
DISASSOCIATE*****/
/*****/
event error_t MLME_DISASSOCIATE.indication(uint32_t
DeviceAddress[], uint8_t DisassociateReason, bool SecurityUse,
uint8_t ACLEntry)
{

    return SUCCESS;
}

event error_t MLME_DISASSOCIATE.confirm(uint8_t status)
{

    return SUCCESS;
}

    event error_t MCPS_DATA.confirm(uint8_t msduHandle, uint8_t
status){

    return SUCCESS;
}
}

```

Con esto termina la aplicación Osciloscopio. A continuación se define un archivo de configuración donde se indican las relaciones de los componentes y de los nuevos componentes de 802.15.4 y la implementación realizada:

```

#include "simpleroutingexample.h"
#include "phy_const.h"
#include "phy_enumerations.h"
#include "mac_const.h"
#include "mac_enumerations.h"
#include "mac_func.h"

```

```

configuration OscilloscopeAppC { }

implementation
{
    components OscilloscopeC;
    components MainC;
    components LedsC;

    components new TimerMilliC() as TimerPrinc;
    components new TimerMilliC() as Timer0;

    components new SensirionSht11C();
    components Mac;
    components SerialActiveMessageC;

    OscilloscopeC.Boot -> MainC;
    OscilloscopeC.RadioControl -> SerialActiveMessageC;
    OscilloscopeC.TimerPrinc -> Timer0;
    OscilloscopeC.Timer0 -> TimerPrinc;
    OscilloscopeC.Read -> SensirionSht11C.Temperature;
    OscilloscopeC.Leds -> LedsC;

    //MAC interfaces

    OscilloscopeC.MLME_START->Mac.MLME_START;

    OscilloscopeC.MLME_GET->Mac.MLME_GET;
    OscilloscopeC.MLME_SET->Mac.MLME_SET;

    OscilloscopeC.MLME_BEACON_NOTIFY->Mac.MLME_BEACON_NOTIFY;
    OscilloscopeC.MLME_GTS->Mac.MLME_GTS;

    OscilloscopeC.MLME_ASSOCIATE->Mac.MLME_ASSOCIATE;
    OscilloscopeC.MLME_DISASSOCIATE->Mac.MLME_DISASSOCIATE;

    OscilloscopeC.MLME_ORPHAN->Mac.MLME_ORPHAN;
    OscilloscopeC.MLME_SYNC->Mac.MLME_SYNC;
    OscilloscopeC.MLME_SYNC_LOSS->Mac.MLME_SYNC_LOSS;
    OscilloscopeC.MLME_RESET->Mac.MLME_RESET;

    OscilloscopeC.MLME_SCAN->Mac.MLME_SCAN;

    OscilloscopeC.MCPS_DATA->Mac.MCPS_DATA;
}

```

Finalmente añadimos los archivos de cabecera con las variables que usará nuestro programa. El fichero `simpleroutingexample.h` contiene información sobre la trama MAC, de los beacons y el identificador del PANCoordinator. Este fichero es común para los dos extremos.

```

enum {
    COORDINATOR = 0x00,
    ROUTER = 0x01,
    END_DEVICE = 0x02
}

```

```

};

#define BEACON_ORDER 3
#define SUPERFRAME_ORDER 3
//the starting channel needs to be different than the existing
coordinator operating channels
#define LOGICAL_CHANNEL 0x15

#define TYPE_DEVICE END_DEVICE
//#define TYPE_DEVICE COORDINATOR

//PAN VARIABLES
#define MAC_PANID 0x1234

```

El fichero Oscilloscope.h contiene nuevamente información del tiempo de muestreo, número de muestras.

```

enum {
    /* Number of readings per message. If you increase this, you
    may have to
    increase the message_t size. */
    NREADINGS = 10,

    /* Default sampling period. */
    DEFAULT_INTERVAL = 128,

};

typedef nx_struct oscilloscope {
    nx_uint16_t version; /* Version of the interval. */
    nx_uint16_t interval; /* Sampling period. */
    nx_uint16_t id; /* Mote id of sending mote. */
    nx_uint16_t count; /* The readings are samples count *
NREADINGS onwards */
    nx_uint16_t readings[NREADINGS];
} oscilloscope_t;

#endif

```

Una vez hecho esto, se compila el código y se carga en el dispositivo TelosB empleando las siguientes sentencias en la consola:

```
> motelist (con esto podremos extraer el puerto serie al que está conectado el dispositivo)
```

```
>make telosb install,2 bsl,NºPuerto Serie – 1
```

3.4 BaseStation en Open-Zigbee

Esta aplicación se encargará de recibir los mensajes del otro dispositivo de nuestra red de sensores inalámbricos, y redirigirlo al PC a través del puerto serie. Para ello se encapsulará el array con los datos recibidos a través de la interfaz vía radio de 802.15.4, y se introducirá en una estructura para ser enviados de la misma manera que hacíamos en TinyOS a través del puerto serie al PC.

Implementación:

Los ficheros correspondientes a ésta implementación se encuentran en la carpeta `\apps\BaseStationHurray\` del directorio raíz.

Primeramente detallamos la implementación realizada. El fichero con la implementación se llama `BaseStationP.nc`.

Lo primero que se hace es cargar las interfaces que van a ser usadas por nuestro programa para el dispositivo TelosB, entre ellas las interfaces MAC de Open Zigbee.

```
#include "AM.h"
#include "Serial.h"
#include <Timer.h>

module BaseStationP @safe() {

    uses interface Boot;
    uses interface Timer<TMilli> as Timer0;

    uses interface SplitControl as SerialControl;

    uses interface AMSend;
    uses interface Receive;
    uses interface Packet as UartPacket;
    uses interface AMPacket as UartAMPacket;

    uses interface Leds;

    //MAC interfaces

    uses interface MLME_START;

    uses interface MLME_GET;
    uses interface MLME_SET;

    uses interface MLME_BEACON_NOTIFY;
    uses interface MLME_GTS;

    uses interface MLME_ASSOCIATE;
    uses interface MLME_DISASSOCIATE;

    uses interface MLME_ORPHAN;

    uses interface MLME_SYNC;
    uses interface MLME_SYNC_LOSS;
```

```

    uses interface MLME_RESET;

    uses interface MLME_SCAN;

    uses interface MCPS_DATA;

}

```

A continuación se definen e inicializan las variables que van a ser usadas por nuestro programa

```

implementation {
    PANDescriptor pan_des;
    uint32_t my_short_address=0x00000000;

    uint32_t SourceMoteAddr[2];
    uint32_t DestinationMote[2];
    uint8_t msdu_payload[28];

    enum {
        UART_QUEUE_LEN = 12,
        RADIO_QUEUE_LEN = 12,
    };

    message_t uartQueueBufs[UART_QUEUE_LEN];
    message_t * mensaje;
    message_t * ONE_NOK uartQueue[UART_QUEUE_LEN];
    uint8_t uartIn, uartOut;
    uint8_t i,j,k;
    bool uartBusy, uartFull;

    message_t radioQueueBufs[RADIO_QUEUE_LEN];
    message_t * ONE_NOK radioQueue[RADIO_QUEUE_LEN];
    uint8_t radioIn, radioOut;
    bool radioBusy, radioFull;
}

```

Se crean las tareas de envío a través de Radio y Uart que luego serán llamadas

```

task void uartSendTask();
task void radioSendTask();

```

Se crean funciones de llamada a Leds para monitorizar el envío de paquetes

```

void dropBlink() {
    call Leds.led2Toggle();
}

void failBlink() {
    call Leds.led2Toggle();
}

```


A continuación se arranca el manejador de la aplicación y arrancamos las componentes que vayamos a usar. Se vacían los arrays de mensajes y se actualizan las variables para señalarlos.

```
event void Boot.booted() {

    i=0;

    for (i = 0; i < UART_QUEUE_LEN; i++)
        uartQueue[i] = &uartQueueBufs[i];
    uartIn = uartOut = 0;
    uartBusy = FALSE;
    uartFull = TRUE;

    for (i = 0; i < RADIO_QUEUE_LEN; i++)
        radioQueue[i] = &radioQueueBufs[i];
    radioIn = radioOut = 0;
    radioBusy = FALSE;
    radioFull = TRUE;
```

Se arranca la componente de la interfaz serial y se definen las direcciones para el envío de mensajes con la interfaz de envío de 802.15.4. Se lanza un temporizador no periódico para 4 segundos.

```
call SerialControl.start();

    DestinationMote[0]=0x00000000;
    DestinationMote[1]=0x00000002;

    SourceMoteAddr[0]=0x00000000;
    SourceMoteAddr[1]=0x00000000;
    my_short_address=0x0000;

    call Timer0.startOneShot(4000);

}
```

Cuando se recoge el evento del temporizador anterior, se inicializan la dirección corta MAC y la PANID que sean variables. Asimismo el dispositivo empieza a enviar balizas (beacons) al resto de dispositivos en la red con el fin de sincronizarse.

```
event void Timer0.fired() {

    uint8_t v_temp[2];

    // Poner la direccion corta MAC variable

    v_temp[0] = (uint8_t)(my_short_address >> 8);
    v_temp[1] = (uint8_t)(my_short_address );
```

```

    call MLME_SET.request(MACSHORTADDRESS,v_temp);

    // Poner el MAC PANID variable

    v_temp[0] = (uint8_t)(MAC_PANID >> 8);
    v_temp[1] = (uint8_t)(MAC_PANID );

    call MLME_SET.request(MACPANID,v_temp);

    // Empieza a enviar Beacons

    call
MLME_START.request(MAC_PANID,LOGICAL_CHANNEL,BEACON_ORDER,SUPERF
RAME_ORDER,1,0,0,0,0);

}

```

Se recogen los distintos eventos para el arranque correcto de las componentes a usar

```

event void SerialControl.startDone(error_t error) {
    if (error == SUCCESS) {
        uartFull = FALSE;
    }
}

event void SerialControl.stopDone(error_t error) {}

uint8_t count = 0;

```

Cuando recibimos un mensaje vía radio de otro dispositivo se lanza este evento.

En dicho evento, MCPS_DATA.indication característico de 802.15.4, se recogen los datos del array de payload (msdu[]) del mensaje recibido y se encapsulan dentro de una estructura message_t la cual será puesta en la cola de envío a través del puerto serie. Acto seguido se lanza la tarea de envío a través del puerto serie (uartSendTask).

```

// RUTINA DE RECEPCION DE MENSAJES

event error_t MCPS_DATA.indication(uint16_t SrcAddrMode,
uint16_t SrcPANId, uint32_t SrcAddr[2], uint16_t DstAddrMode,
uint16_t DestPANId, uint32_t DstAddr[2], uint16_t
msduLength,uint8_t msdu[28],uint16_t mpduLinkQuality, uint16_t
SecurityUse, uint16_t ACLEntry)
{

    call Leds.led0Toggle();

    j = 0;

    for (j=0; j<29 ; j++){

        mensaje->data[j]=msdu[j];
    }
}

```

```
    }  
    atomic {  
        if (!uartFull)  
        {  
            message_t *ret = mensaje;  
  
            ret = uartQueue[uartIn];  
            uartQueue[uartIn] = mensaje;  
  
            uartIn = (uartIn + 1) % UART_QUEUE_LEN;  
  
            if (uartIn == uartOut)  
                uartFull = TRUE;  
  
            if (!uartBusy)  
            {  
                post uartSendTask();  
                uartBusy = TRUE;  
            }  
        }  
        else  
            dropBlink();  
    }  
  
    return SUCCESS;  
}
```

A continuación nuestra aplicación recoge la implementación de dicha tarea de envío a través del puerto serie al PC.

```
uint8_t tmpLen;

// #####
// #####  RUTINA DE ENVIO A UART  #####
// #####

task void uartSendTask() {

    uint8_t len;

    // am_id_t id;
    // am_addr_t addr, src;

    message_t* msg;
    atomic
    if (uartIn == uartOut && !uartFull)
    {
        uartBusy = FALSE;
        return;
    }

    msg = uartQueue[uartOut];
    len = call UartPacket.payloadLength(msg);
    call UartPacket.clear(msg);

    if(call AMSend.send(AM_BROADCAST_ADDR, uartQueue[uartOut],
len) == SUCCESS){
        call Leds.led1Toggle();
    }

    else
    {
        failBlink();
        post uartSendTask();
    }
}

event void AMSend.sendDone(message_t* msg, error_t error) {
    if (error != SUCCESS)
        failBlink();
    else
        atomic
        if (msg == uartQueue[uartOut])
        {
            if (++uartOut >= UART_QUEUE_LEN)
                uartOut = 0;
            if (uartFull)
                uartFull = FALSE;
        }
        post uartSendTask();
}
```

Este evento se produce cuando se recibe a través del puerto serie algún mensaje para enviar vía interfaz radio para los demás dispositivos conectados a nuestra red. Cuando ello se produce se lanza la tarea de envío a través de radio (radioSendTask).

```
// #####  
// ### RUTINA DE RECEPCION DE UART ###  
// #####  
  
event message_t* Receive.receive(message_t *msg,  
                                void *payload,  
                                uint8_t len) {  
  
    message_t *ret = msg;  
    bool reflectToken = FALSE;  
  
    atomic  
    if (!radioFull)  
    {  
        reflectToken = TRUE;  
        ret = radioQueue[radioIn];  
        radioQueue[radioIn] = msg;  
        if (++radioIn >= RADIO_QUEUE_LEN)  
            radioIn = 0;  
        if (radioIn == radioOut)  
            radioFull = TRUE;  
  
        if (!radioBusy)  
        {  
            post radioSendTask();  
            radioBusy = TRUE;  
        }  
    }  
    else  
        dropBlink();  
  
    if (reflectToken) {  
        //call UartTokenReceive.ReflectToken(Token);  
    }  
  
    return ret;  
}
```

Finalmente implementamos dicha tarea de envío a través de la interfaz de radio de nuestro dispositivo. Para ello, usaremos la interfaz que nos provee 802.15.4 de envío de datos (MCPS_DATA.request)

```

// #####
// ###  RUTINA DE ENVIO POR RADIO  ###
// #####

task void radioSendTask() {

    message_t* msg;

    atomic
    if (radioIn == radioOut && !radioFull)
    {
        radioBusy = FALSE;
        return;
    }

    msg = radioQueue[radioOut];

    k=0;

    msdu_payload[0]=0x00;

    for(k=0 ; k<29 ; k++){
    msdu_payload[k] = radioQueue[radioOut]->data[k];
    }

    call MCPS_DATA.request(SHORT_ADDRESS, MAC_PANID,
SourceMoteAddr, SHORT_ADDRESS, MAC_PANID, DestinationMote, 1,
msdu_payload,1,set_txoptions(0,0,0,0));

    atomic
    if (msg == radioQueue[radioOut])
    {
        if (++radioOut >= RADIO_QUEUE_LEN)
            radioOut = 0;
        if (radioFull)
            radioFull = FALSE;
    }

    post radioSendTask();
}

/*****
*****/
/*****MLME-
SCAN*****/
/*****
*****/

event error_t MLME_SCAN.confirm(uint8_t status,uint8_t ScanType,
uint32_t UnscannedChannels, uint8_t ResultListSize, uint8_t
EnergyDetectList[], SCAN_PANDescriptor PANDescriptorList[])

```

```

{
    return SUCCESS;
}
/*****
*****/
/*****MLME-
ORPHAN*****/
/*****
*****/
event error_t MLME_ORPHAN.indication(uint32_t OrphanAddress[1],
uint8_t SecurityUse, uint8_t ACLEntry)
{
    return SUCCESS;
}
/*****
*****/
/*****MLME-
RESET*****/
/*****
*****/
event error_t MLME_RESET.confirm(uint8_t status)
{
    return SUCCESS;
}
/*****
*****/
/*****MLME-SYNC-
LOSS*****/
/*****
*****/
event error_t MLME_SYNC_LOSS.indication(uint8_t LossReason)
{
    return SUCCESS;
}
/*****
*****/
/*****MLME-
GTS*****/
/*****
*****/
event error_t MLME_GTS.confirm(uint8_t GTSCharacteristics,
uint8_t status)
{

```

```

    return SUCCESS;
}

```

```

event error_t MLME_GTS.indication(uint16_t DevAddress, uint8_t
GTSCharacteristics, bool SecurityUse, uint8_t ACLEntry)
{
    return SUCCESS;
}

/*****
*****/
/*****MLME-BEACON
NOTIFY*****/
/*****
*****/

event error_t MLME_BEACON_NOTIFY.indication(uint8_t
BSN,PANDescriptor pan_descriptor, uint8_t PenAddrSpec, uint8_t
AddrList, uint8_t sduLength, uint8_t sdu[])
{
    return SUCCESS;
}

/*****
*****/
/*****MLME-
START*****/
/*****
*****/

event error_t MLME_START.confirm(uint8_t status)
{
    return SUCCESS;
}

/*****
*****/
/*****MLME-SET
*****/
/*****
*****/

event error_t MLME_SET.confirm(uint8_t status,uint8_t
PIBAttribute)
{
    return SUCCESS;
}

/*****
*****/
/*****MLME-GET
*****/
/*****
*****/

event error_t MLME_GET.confirm(uint8_t status,uint8_t
PIBAttribute, uint8_t PIBAttributeValue[])
{

```



```

return SUCCESS;
}

/*****
*****/
/*****MLME-
ASSOCIATE*****/
/*****
*****/

event error_t MLME_ASSOCIATE.indication(uint32_t
DeviceAddress[], uint8_t CapabilityInformation, bool
SecurityUse, uint8_t ACLEntry)
{

    return SUCCESS;
}

event error_t MLME_ASSOCIATE.confirm(uint16_t AssocShortAddress,
uint8_t status)
{

    return SUCCESS;
}

/*****
*****/
/*****MLME-
DISASSOCIATE*****/
/*****
*****/

event error_t MLME_DISASSOCIATE.indication(uint32_t
DeviceAddress[], uint8_t DisassociateReason, bool SecurityUse,
uint8_t ACLEntry)
{

    return SUCCESS;
}

event error_t MLME_DISASSOCIATE.confirm(uint8_t status)
{

    return SUCCESS;
}

/*****
*****/
/*****
*****/
/*****
*****
MCPS EVENTS
*****/
/*****
*****/
/*****
*****/

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

```



```

BaseStationP.MLME_START -> Mac.MLME_START;

BaseStationP.MLME_GET ->Mac.MLME_GET;
BaseStationP.MLME_SET ->Mac.MLME_SET;

BaseStationP.MLME_BEACON_NOTIFY ->Mac.MLME_BEACON_NOTIFY;
BaseStationP.MLME_GTS -> Mac.MLME_GTS;
BaseStationP.MLME_ASSOCIATE->Mac.MLME_ASSOCIATE;
BaseStationP.MLME_DISASSOCIATE->Mac.MLME_DISASSOCIATE;
BaseStationP.MLME_ORPHAN->Mac.MLME_ORPHAN;
BaseStationP.MLME_SYNC->Mac.MLME_SYNC;
BaseStationP.MLME_SYNC_LOSS->Mac.MLME_SYNC_LOSS;
BaseStationP.MLME_RESET->Mac.MLME_RESET;
BaseStationP.MLME_SCAN->Mac.MLME_SCAN;
BaseStationP.MCPS_DATA->Mac.MCPS_DATA;

}

```

Creamos un archivo de cabecera, los valores han de ser iguales que los valores del archivo de cabecera de la otra aplicación Oscilloscope llamado BaseStation.h

```

enum {
    COORDINATOR = 0x00,
    ROUTER =0x01,
    END_DEVICE = 0x02,
    AM_TEST_SERIAL_MSG = 9
};

#define BEACON_ORDER 3
#define SUPERFRAME_ORDER 3
//the starting channel needs to be diferrent that the existent
coordinator operating channels
#define LOGICAL_CHANNEL 0x15

//#define TYPE_DEVICE END_DEVICE
#define TYPE_DEVICE COORDINATOR

//PAN VARIABLES
#define MAC_PANID 0x1234

```

Finalmente compilaremos nuestro programa y lo cargaremos en nuestro dispositivo utilizando para ello las siguientes en nuestra consola de comandos:

> *motelist* (con ésto podremos extraer el puerto serie al que está conectado el dispositivo)

>*make telosb install,1 bsl,NºPuerto Serie – 1*

Conclusiones:

Una vez compilados y arrancados ambos dispositivos, el nodo Osciloscopio que toma muestras de temperatura empezará a enviar mensajes que serán recogidos por el otro nodo BaseStation. Dichos mensajes serán reencapsulados y puestos en cola para ser transmitidos vía puerto serie al PC.

Capítulo 5.- Apartado de Resultados del proyecto

Para comprobar el correcto funcionamiento del modelo planteado de envío y recepción de muestras de temperatura y su posterior visualización por pantalla, se ha procedido a realizar varias modificaciones a la aplicación Java Osciloscopio para extraer conclusiones, recoger muestras en archivos de texto, calcular el número máximo de muestras a enviar, y el menor y mayor tiempo de intervalo de muestreo.

Para ello, se ha procedido a modificar la aplicación Java Oscilloscope que permite, desde el extremo PC, visualizar muestras recibidas por BaseStation del dispositivo Osciloscopio.

La aplicación Java Osciloscopio cuenta con varias clases que procedemos a detallar:

- **Colorcelleditor.java**
- **Graph.java**
- **Window.java**

Todas ellas son necesarias para la creación de la interfaz de usuario gráfica y de dibujado de la monitorización de muestras por pantalla.

- **Constants.java**
- **OscilloscopeMsg.java**

Estas dos clases son creadas por mig. En ellas quedan guardadas para el uso del programa principal Java Osciloscopio los valores de número de muestras por trama, y el intervalo para recoger las muestras. Cada vez que cambiemos a nuestro programa el tiempo de intervalo de muestreo, o el número de muestras a tomar por trama, deberemos recompilar en Java haciendo un “make” para que mig vuelva a generarlas.

- **Data.java** (clase para el manejo de datos)
- **Node.java** (clase que almacena los datos recibidos por un mote)

Éstas dos clases son las encargadas del manejo y almacenamiento de los mensajes recibidos del mote Osciloscopio de nuestra red, y el almacenamiento de las muestras de temperatura que van dentro de la carga de datos de los mismos (*payload*)

Nos centraremos en la clase Node.java, ya que es en dicha clase donde se almacenan las muestras recibidas en los mensajes *message_t* que recibe BaseStation. Dentro de la clase actualiza constantemente el array de muestras recibidas, *data []*, para luego ser

usadas por las clases del programa Java que dibujarán la variación de temperatura en pantalla.

La modificación desarrollada en el proyecto para Node.java consiste en leer dicho array de enteros (int) y convertir a un String de Java *tokenizado* (consiste en separar con espacios las muestras) las muestras recibidas por nuestra aplicación BaseStation en cada uno de los mensajes. Todas las muestras quedarán guardadas en un archivo de texto haciendo uso de las funciones que nos provee el paquete Java.IO (entrada-salida).

Podemos verlo claramente en el código del programa a continuación:

```
/**
 * Clase que contiene los datos recibidos por un dispositivo
 */

import java.io.*;

class Node {

    . . .
```

- Aquí se hace la llamada desde la función que actualiza el array de enteros a otra función que guarde en un archivo. El array de enteros data [] se pasa por la función.

```
. . .

void update(int messageId, int readings[]) {
    int start = messageId * Constants.NREADINGS;
    setEnd(start, start + Constants.NREADINGS);
    for (int i = 0; i < readings.length; i++)
        data[start - dataStart + i] = readings[i];

    escribirFichero(data);
}

. . .
```

- Aquí se desarrolla la función de guardado en un fichero de las muestras. Primero recoge de la función el array de enteros data[], lee datos del array de enteros donde se guardan las muestras, luego convierte a un String de Java todas las

muestras y *tokenizado* de cada muestra, y finalmente se envía a un archivo externo .txt.

```
• • •
    void escribirFichero(int datos[]){

        StringBuffer buf = new StringBuffer();
        buf.append(datos[0]);

        for (int i = 1; i < datos.length; buf.append("
").append(datos[i++]));
        String datosString = buf.toString();

        //FileOutputStream f = new FileOutputStream (
        "muestras.txt");
        //ObjectOutputStream out = new ObjectOutputStream(f);
        //out.writeObject(data);
        //out.flush();
        //out.close();

        try{

            FileWriter muestras = null;

            muestras= new FileWriter("muestras.txt");
            PrintWriter out = new PrintWriter(muestras);
            out.print(datosString);
            out.close();
        }

        catch (Exception e) {
            e.printStackTrace();
        }

        //catch (IOException e) {

        //System.out.println(e);

        }
    }
}
```

Una vez hecho todo esto, debemos recompilar nuestra aplicación Java Osciloscopio.

Para ello dentro de la ruta de nuestro programa escribiremos:

```
/tos/apps/Oscilloscope/Java/ > "make"
```

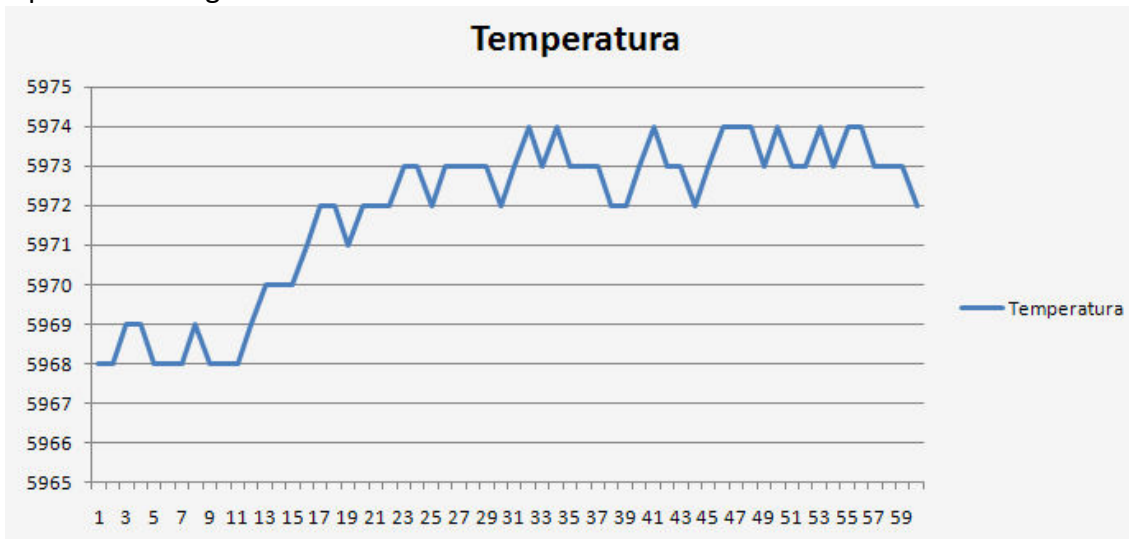
Una vez realizado “make” ya tenemos lista nuestra aplicación con las modificaciones necesarias para recoger las muestras en un fichero de texto. Al hacer “make” reconfiguramos el fichero contenedor .jar con las distintas clases necesarias para correr la aplicación.

Ahora para testearlo sólo tenemos que arrancar SerialForwarder de la misma manera que lo hicimos originalmente:

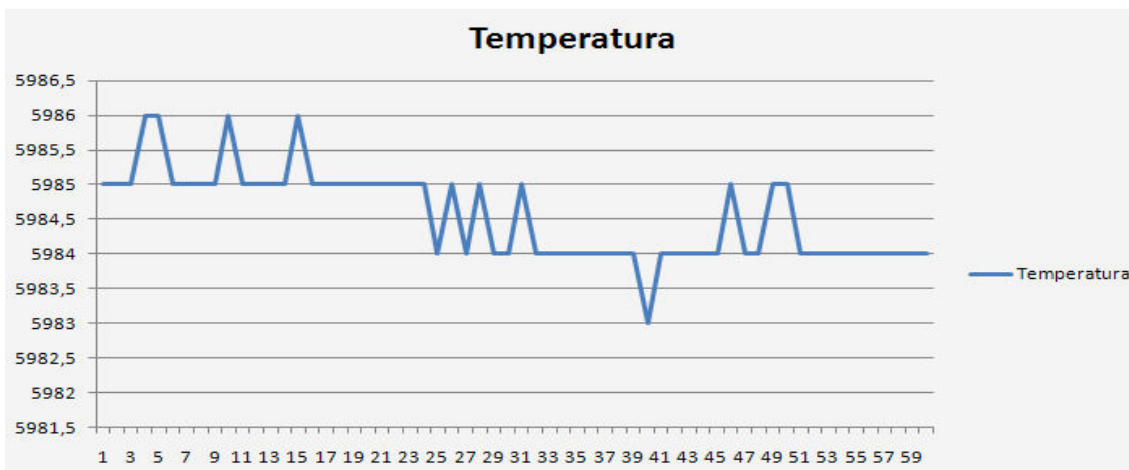
```
>java net.tinyos.sf.SerialForwarder -comm  
serial@/dev/ttyUSBx:telosb &
```

A continuación abrimos la aplicación Java Osciloscopio, además de monitorizar las muestras de temperatura que vayan llegando con una gráfica, se irán guardando en un archivo “muestras.txt” las muestras recogidas.

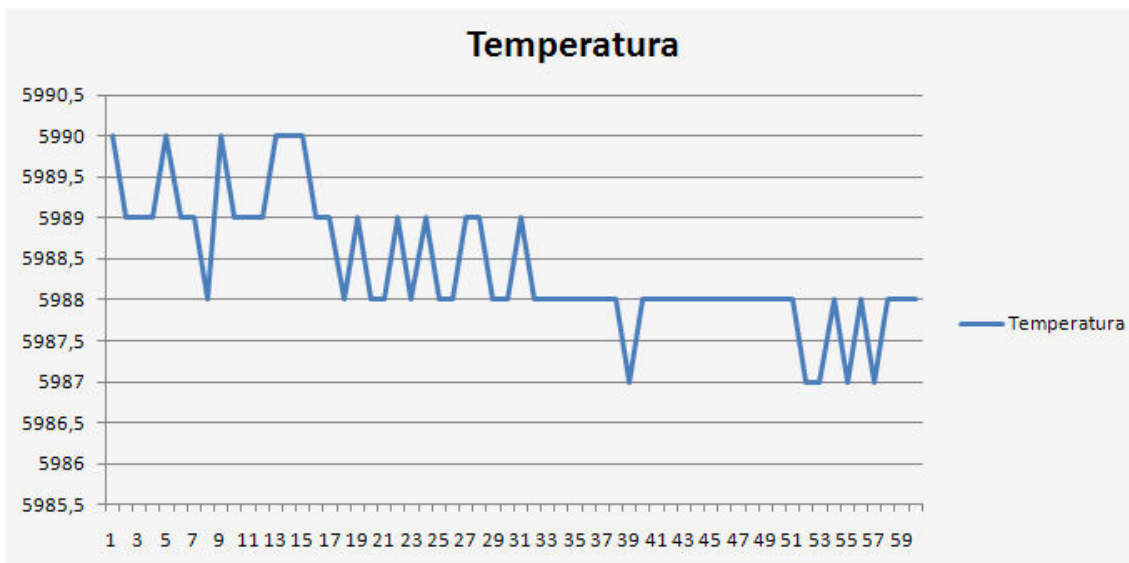
Una vez hechas las modificaciones tanto en Open Zigbee como en TinyOS comprobamos los resultados de ambos con varias tomas de muestras y su posterior representación gráfica con una tabla Excel.



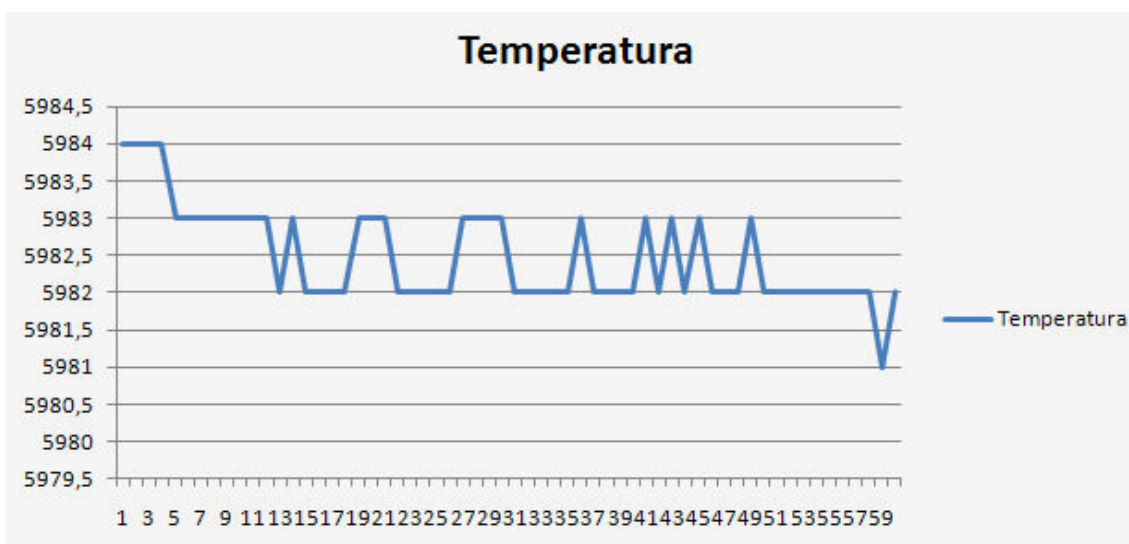
10 muestras con intervalo 256mseg. en TinyOS



10 muestras con intervalo 256mseg. en Open Zigbee



10 muestras con intervalo de 64 mseg. en TinyOS



10 muestras con intervalo de 64 mseg. en Open Zigbee

Podemos comprobar en ambas gráficas que el funcionamiento del sensor en Open Zigbee es igual al que tenemos en TinyOS.

Por defecto, y como hemos dicho anteriormente, nuestro programa Osciloscopio está configurado por defecto para enviar 10 muestras cada intervalo de 256 milisegundos.

Con el fin de testear el tamaño máximo de paquete que se puede enviar entre motes, relacionado con el máximo de muestras leídas por paquete, y el intervalo mínimo de muestreo y máximo, se modifica el valor de dichas constantes.

Ambos valores se pueden modificar en el archivo de cabecera:

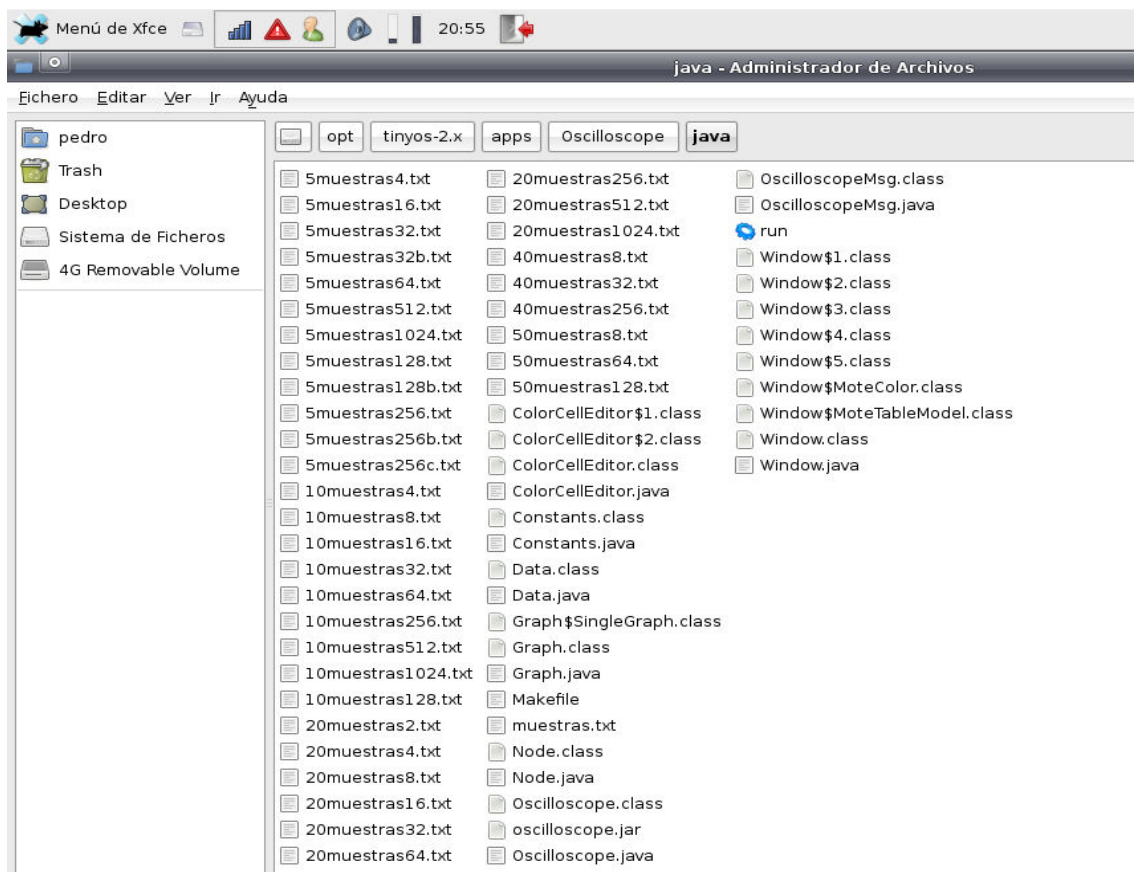
/tos/apps/Oscilloscope/Oscilloscope.h

Debemos recompilar la aplicación Osciloscopio y volver a cargársela al mote, y también deberemos volver a recompilar la aplicación Java Osciloscopio haciendo “make” en su ruta:

/tos/apps/Oscilloscope/java/ > make

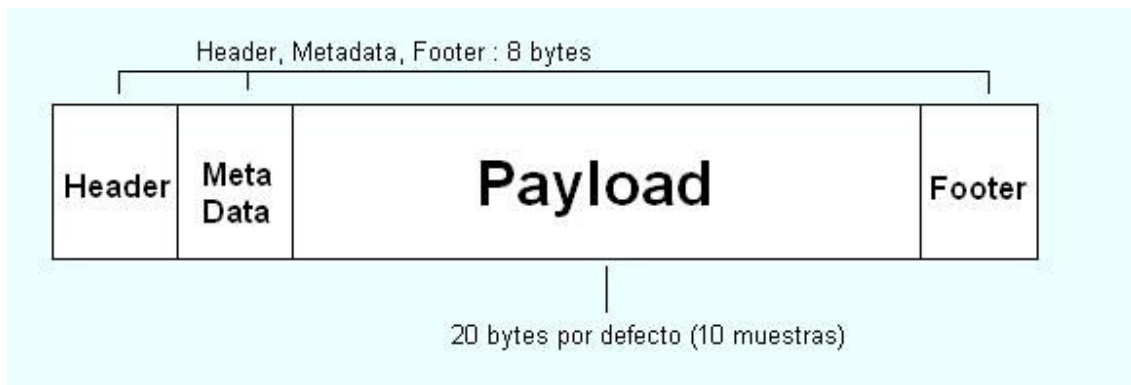
Una vez modificado el intervalo con un rango desde 1024 milisegundos, hasta los 4 milisegundos, nuestra aplicación no tiene ningún problema para tomar lecturas, transmitir las, y almacenarse estas en el fichero de texto.

Asimismo se ha operado con lecturas desde 1 hasta las 10 lecturas por defecto, sin encontrar ningún problema aparente.



Composición de varios archivos guardados durante el testeó

Ahora bien, encontraremos problemas cuando queramos transmitir mas muestras de temperatura por paquete, y deberemos hacer modificaciones a nuestro sistema.



Trama *message_t* con tamaño 28 bytes, 10 muestras de temperatura

La longitud de trama por defecto para los mensajes transmitidos en nuestra red, que son *message_t* es de 28 bytes, según el valor por defecto dado en el archivo de cabecera que configura los *message_t* que se encuentra en:

/tos/types/message.h

Un paquete enviado con 10 muestras de temperatura ocuparía: 20 bytes (2 bytes por cada muestra, que son igual a los 16 –bits) y 8 bytes de campos de la propia trama *message_t*, que son *metadata*, *header* y *footer*.

Si queremos enviar un paquete con más muestras debemos modificar el archivo de cabecera *message.h*, del cual hemos dicho ya su ruta, y la variable a modificar será:

TOSH_DATA_LENGTH = 28

que es el tamaño de trama de los mensajes *message_t*.

Como nota importante, hay que tener en cuenta que no solo habrá que modificar el valor de *TOSH_DATA_LENGTH* en este archivo común, sino también en el del chip de radio que usamos en nuestros dispositivos TelosB, en nuestro caso el CC2420.

Se encuentra el archivo en:

/tos/chips/cc2420/CC2420.h

Se modifica la misma constante *TOSH_DATA_LENGTH* en dicho archivo, de hecho si no lo hacemos, el tamaño que usará nuestra aplicación será este, y si el valor es menor que el necesitado para transmitir el paquete, tendremos problema de transmisión de mensajes.

Si queremos enviar un paquete con 15 muestras, deberemos aumentar en 10 bytes el tamaño de la trama *message_t*, y así sucesivamente, si queremos enviar un paquete con mas muestras tomadas de temperatura.

De no optimizar el tamaño de *message_t* la aplicación Java Osciloscopio nos informará de que el tamaño mínimo de mensaje que espera recibir para analizar de BaseStation no coincide, y se ha producido un error.

Una vez hecha la modificación del tamaño de *message_t* que contenga a las muestras enviadas que deseemos, empezamos a testear nuestra aplicación con tamaños de paquete mayores, con envío de más de 10 muestras por paquete.

Se testea con 20, 30, 40, 50 y más muestras.

Más allá de 50 muestras nuestra aplicación BaseStation no se sincroniza con Osciloscopio y deja de funcionar la recepción y monitorización de muestras de temperatura en la aplicación Java Osciloscopio.

Así pues podemos decir que el límite máximo de tamaño de nuestro paquete a enviar será de:

108 bytes : 100 bytes correspondientes a 50 muestras de 2 bytes cada una y 8 bytes de datos de *header, metadata, y footer*.

No se han encontrado limitaciones en el intervalo de tiempo en el que tomar las muestras, tomando como rango desde los 4 milisegundos hasta los 1024.

Conclusiones y trabajos futuros.

Durante el desarrollo de este Proyecto Fin de Carrera se han obtenido las siguientes conclusiones funcionales, útiles para futuros desarrollos:

El sistema operativo TinyOS nos provee las herramientas para poder crear aplicaciones que exploten las bondades de las redes de sensores inalámbricos de baja potencia (LR-WPAN). Dichas herramientas están basadas en componentes de software de las que haremos uso como librerías haciendo llamadas a las interfaces que nos proveen. Ellas están bien diferenciadas dependiendo de la arquitectura hardware que estemos usando.

Entonces comprobamos que crear aplicaciones basadas en TinyOS es un proceso laborioso debido al uso del nuevo lenguaje de programación nesC, con sus restricciones. Como apoyo nos sirven la gran cantidad de librerías proporcionadas por el equipo de desarrollo de TinyOS.

El uso de la distribución de Linux XubunTOS, con TinyOS preinstalado nos facilita más aún el desarrollo de aplicaciones basadas en TinyOS, ya que nos instala todo el software que podamos requerir para desarrollar aplicaciones.

Asimismo, si queremos usar como entorno de trabajo Cygwin deberemos tener algunas consideraciones a la hora de compilar nuestras aplicaciones por consola, o por ejemplo, cargando las variables de entorno que nos van a permitir compilar nuestros programas. Cygwin nos permitirá programar y trabajar bajo Windows con nuestros dispositivos inalámbricos, pero para hacerlo funcionar correctamente, tendremos que tener en cuenta que su instalación y puesta en marcha es más compleja que XubuntuTOS.

Se ha llegado a la conclusión de que TinyOS no hace uso del protocolo 802.15.4 y de todas sus características, y que para usar el mismo bajo TinyOS hemos instalado una implementación Zigbee de código abierto y libre llamada Open-Zigbee.

Con Open-Zigbee se han dado los primeros pasos conociendo el protocolo, haciendo envío y recepción de datos a través de sus interfaces de envío de paquetes. Las aplicaciones que hemos creado para Open Zigbee nos permiten configurar de una manera más abierta las características de la red que en TinyOS, por lo que hacemos un uso más eficiente de nuestros dispositivos.

En cuanto a lo personal, este proyecto me ha hecho descubrir nuevas tecnologías inalámbricas y protocolos de comunicaciones que desconocía. Además la comprensión de nesC, redes de sensores usando TinyOS y 802.15.4 puede llegar a ser útil en mi desarrollo laboral.

En cuanto a posibles líneas futuras de este Proyecto Fin de Carrera:

- Este trabajo debe permitir el desarrollo de aplicaciones optimizadas en el muestreo de temperatura-humedad en redes de dispositivos inalámbricos de baja potencia.
- Se puede plantear crear aplicaciones más complejas, con redes con más dispositivos tomando muestras de temperatura-humedad y enrutar los mensajes a un coordinador dentro de la red.

Bibliografía y Referencias

- [1] TinyOS - <http://www.tinyos.net/>
- [2] Mote TelosB - <http://www.xbow.com/Products/productdetails.aspx?sid=252>
- [3] CrossBow - <http://www.xbow.com>
- [4]Tipos de modulaciones en el 802.15.4 - <http://es.wikipedia.org/wiki/DSSS>

Crossbow Getting Started Guide –

http://www.xbow.com/Support/Support_pdf_files/Getting_Started_Guide.pdf

RF Transmisores- <http://www.ti.com/>

- Mote TelosB Datasheet - [http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Data sheet.pdf](http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Data_sheet.pdf)
- Sensirion SHT-11 Datasheet - http://www.sensirion.com/en/pdf/product_information/Datasheet-humidity-sensor-SHT1x.pdf
- 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver – CC2420 - <http://focus.ti.com/lit/ds/symlink/cc2420.pdf>
- TinyOS Tutorials - http://docs.tinyos.net/index.php/TinyOS_Tutorials
- Cygwin - <http://www.cygwin.com/>
- ZigBee/IEEE 802.15.4 Summary - <http://www.sinemergen.com/zigbee.pdf>
- Especificación de la estructura de trama del 802.15.4 http://www.chipcon.com/index.cfm?kat_id=1&action=faq&faq_id=3
- MSP430 Microcontrollers - <http://focus.ti.com/lit/sg/slab034q/slab034q.pdf>
- An IEEE 802.15.4 protocol implementation(in nesC/TinyOS): Reference Guide v1.2 - André CUNHA, Mário ALVES,..

<http://open->

zb.net/publications/HURRAY_TR_061106_An_IEEE_802.15.4_protocol_implementation%20_in_nesCTinyOS_%20Reference_Guide_v1.2.pdf

- "Low-Rate Wireless Personal Area Networks (LR-WPANs)", Enabling Wireless Sensors with IEEE 802.15.4 Jose A Gutierrez, Edgar H. Callaway Jr., Raymond L. Barrett Jr. 2nd Edition.
- D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, D. Culler, "The nesC Language:A Holistic Approach to Networked Embedded Systems", in PLDI'03.
- http://www.it.uc3m.es/~jgr/publicaciones/Tesis_Jaime.pdf

- IEEE 802.15 WPAN™ Task Group 4 (TG4). Abril/2006:
<http://www.ieee802.org/15/pub/TG4.html>
- MAYNÉ, Jordi. IEEE 802.15.4 y Zigbee. SILICA. Octubre/2004. Última consulta:
Marzo/2006. Disponible en:
http://www.bairesrobotics.com.ar/data/IEEE_ZIGBEE_SILICA.pdf

Apéndice A

Instalacion de TinyOS bajo Windows (Cygwin)

Este tutorial está traducido y extraido de la página web:

<http://tiny-os.blogspot.com/2007/03/tinyos-1x-windows-installation-method-1.html>

Configuración del entorno

TinyOS requiere Cygwin y un entorno de desarrollo Java. Sigue estos pasos para configurarlo.

- 1.- Descargar e instalar Cygwin. Seleccionar los paquetes cvs, gcc, gdb, openssh, perl, rpm y vim en la instalación. Todos los de programación.
- 2.- Descarga e instala el último Java JDK.
- 3.- (Opcional) Descargar e instalar el paquete javax.comm de Sun.

Instalando TinyOS

Las siguientes instrucciones descargarán TinyOS. De acuerdo a que puede contener código no estable, te dará la versión más reciente. Comprobando del CVS puedes siempre alternar la versión de TinyOS o permanecer sincronizado con el código más reciente.

1.- Descarga los siguientes RPM dentro de una carpeta temporal:

- Tinyos-tools-1.2.2-1.cygwin.i386.rpm
- Nesc-1.2.7^a-1.cygwin.i386.rpm
- Make-3.80tinyos-1.cygwin.i386.rpm

2.- Si estás usando Windows Vista, siempre arranca Cygwin en modo “Administrador”.

3.- En Cygwin, ve al directorio temporal donde guardaste los RPM, e instalalos usando:

- `Rpm --ignoreos -ivh *.rpm`

4.- En Cygwin, usa CVS para descargar los TinyOS source files.

(El segundo comando te pedirá un password, pulsar ENTER)

- `cd /opt`

- `cvs -d:pserver:anonymous@tinynos.cvs.sourceforge.net:/cvsroot/tinynoslogin`
 - `cvs -z3 -d:pserver:anonymous@tinynos.cvs.sourceforge.net:/cvsroot/tinynos co -P tinynos-1.x tinynos-2.x`
- 5.- Guardar el fichero Makefile (en la web) en `/opt/tinynos-1.x/tools/make`
- 6.- Guardar `washu.sh` y `tinynos.sh` en `/etc/profile.d`
- 7.- Guardar `locate-jre` en `/usr/local/bin`
- 8.- Compilar el código Java
- `cd /opt/tinynos-1.x/tools/java`
 - `make`
- 9.- Configurar la librería JNI de Java
- `cd /opt/tinynos-1.x/tools/java/jni`
 - `make install`

Instalando soporte de Plataforma

TinyOS trabaja sobre varias arquitecturas de hardware. Aquí encuentras instrucciones de como añadir soporte para Mica2 y TelosB motes.

MSP430 Tools para soporte para familia Telos

- 1.- Si estas usando dispositivos TelosB, descarga e instala el driver FTDI VirtualCom driver para los motes TelosB. Cuando conectas los motes TelosB, Windows buscará dicho driver. Apunta a la carpeta donde guardas el driver para reconocer el dispositivo conectado.
- 2.- Descarga los siguientes RPM en una carpeta temporal:
- `mmsp430tools-binutils-2.16-20050607.cygwin.i386.rpm`
 - `mmsp430tools-base-0.1-20050607.cygwin.i386.rpm`
 - `mmsp430tools-gcc-3.2.3-20050607.cygwin.i386.rpm`
 - `mmsp430tools-libc-20050308cvs-20050608.cygwin.i386.rpm`
- 3.- Abre Cygwin y ve a la carpeta temporal e instala de la siguiente forma:
- `rpm --ignoreos --nodeps --ivh *.rpm`

Testeo de la Instalación

Una vez terminado todos estos pasos la instalación de TinyOS debe haber quedado lista para ser usada. La instalación de TinyOS 2.0.2 se hace de manera igual a la que aquí presentamos, aunque debemos tener en cuenta cambiar las variables de entorno a la hora de arrancar Cygwin.

