PROYECTO FIN DE CARRERA

SISTEMA DE MONITORIZACIÓN BASADO EN EL TELÉFONO MÓVIL NOKIA 6131 NFC PARA UNA RED DE SENSORES INALÁMBRICA ZIGBEE

Dpto. de Ingeniería Electrónica de la Escuela Superior de Ingenieros de Sevilla
y Dpto. de Ingeniería Biofísica y Electrónica (DIBE) de la Universidad de Génova

Proyecto fin de carrera de:

D. Mariano Gallardo Magaña

Dirigido por:

Dr. Federico José Barrero García

Titulación:

Ingeniería en Automática y Electrónica Industrial

Mayo, 2009

ÍNDICE

2. INTRODUCCIÓN Y ANTECEDENTES

En los años 90, las redes han revolucionado la forma en la que las personas y las organizaciones intercambian información y coordinan sus actividades. En ésta década seremos testigos de otra revolución; una nueva tecnología permitirá la observación y el control del mundo físico. Los últimos avances tecnológicos han hecho realidad el desarrollo de unos mecanismos distribuidos, diminutos, baratos y de bajo consumo, que, además, son capaces tanto de procesar información localmente como de comunicarse de forma inalámbrica. La disponibilidad de microsensores y comunicaciones inalámbricas permitirá desarrollar redes de sensores/actuadores para un amplio rango de aplicaciones.

Esto conllevará un necesario desarrollo de modelos físicos, los cuales requieren un análisis y monitorización de datos efectivo y funcional. Un segundo reto a superar es la variabilidad de este nuevo entorno. Mientras un buen sistema distribuido se desarrolla con la fiabilidad como elemento básico, estas nuevas aplicaciones presentan un nivel de aleatoriedad más allá de lo común.

Pero la idea dominante radica en las restricciones impuestas por los sistemas en estado inactivo. Estos sistemas deben ser de bajo consumo y larga duración; tanto cuando operan como cuando permanecen a la espera.

El motivo del éxito de este tipo de redes de sensores se debe a sus especiales características físicas. A los nodos de las WSN se les imponen unas restricciones de consumo severas. El motivo de la imposición de estas restricciones es la necesidad de que los nodos sean capaces de operar, por sí mismos, durante periodos largos de tiempo, en lugares donde las fuentes de alimentación son si no inexistentes, de baja potencia. El tamaño es otra restricción que cada vez se hace más necesaria para la mayoría de las aplicaciones, de manera que las tarjetas o nodos que forman las WSN son cada vez de menor tamaño.

Desde el punto de vista del software, para la realización de las aplicaciones para WSN, la Universidad de Berkeley e Intel han desarrollado una plataforma específica para este tipo de sistemas, que tiene en cuenta las restricciones de los nodos. En particular se ha desarrollado un sistema operativo, llamado TinyOS, cuya característica principal reside en que al ser modular resulta ideal para instalarse en sistemas con restricciones de memoria. Se desarrolló también un lenguaje de programación, llamado nesC, de sintaxis muy parecida a C, basado en componentes, y a partir del cual se rediseñó una primera versión de TinyOS de modo que actualmente está íntegramente implementado sobre nesC. Tanto nesC como TinyOS están basados en componentes e interfaces bi-direccionales. Además, actualmente, Berkeley e Intel han desarrollado diversas aplicaciones a modo de ejemplo, simuladores de ejecución, y varias universidades internacionales están dedicando esfuerzos al desarrollo de aplicaciones usando esta emergente tecnología.

2.1 Redes de Sensores Inalámbricas

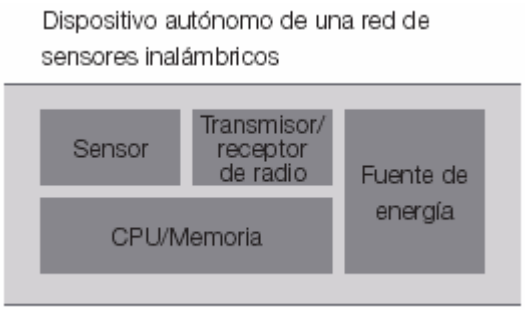
Una red de sensores inalámbrica está formada por nodos. Cada nodo de la red consta de un dispositivo con microcontrolador, sensores y transmisor/receptor, y forma una red con muchos otros nodos, también llamados motas o sensores. Por otra parte, un sensor es capaz de procesar una limitada cantidad de datos. Pero cuando coordinamos la información entre un importante número de nodos, éstos tienen la habilidad de medir un medio físico dado con gran detalle.

Con todo esto, una red de sensores puede ser descrita como un grupo de motas que se coordinan para llevar a cabo una aplicación especifica. Al contrario que las redes tradicionales, las redes de sensores llevarán con más precisión sus tareas dependiendo de lo denso que sea el despliegue y lo coordinadas que estén.

En los últimos años, las redes de sensores han estado formadas por un pequeño número de nodos que estaban conectados por cable a una estación central de procesado de datos. Hoy en día, sin embargo, nos centramos más en redes de sensores distribuidas e inalámbricas. Pero, por qué distribuidas e inalámbricas: cuando la localización de un fenómeno físico es desconocida, este modelo permite que los sensores estén mucho más cerca del evento de lo que estaría un único sensor.

Además, en muchos casos, se requieren muchos sensores para evitar obstáculos físicos que obstruyan o corten la línea de comunicación. El medio que va a ser monitorizado no tiene una infraestructura, ni para el suministro energético, ni para la comunicación. Por ello, es necesario que los nodos funcionen con pequeñas fuentes de energía y que se comuniquen por medio de canales inalámbricos.

Otro requisito para las redes de sensores será la capacidad de procesamiento distribuido. Esto es necesario porque, siendo la comunicación el principal consumidor de energía, un sistema distribuido significará que algunos sensores necesitarán comunicarse a través de largas distancias, lo que se traducirá en mayor consumo. Por ello, es una buena idea el procesar localmente la mayor cantidad de energía, para minimizar el número de bits transmitidos.



Tipos de motas

Las Wireless Sensor Networks, tienen una corta historia, a pesar de ello, ya tenemos a varios fabricantes trabajando en esta tecnología:

CROSSBOW: Especializada en el mundo de los sensores, es una empresa que desarrolla plataformas hardware y software que dan soluciones para las redes de sensores inalámbricas. Entre sus productos encontramos las plataformas Mica, Mica2, Micaz, Mica2dot, telos y telosb.

MOTEIV: Joseph Polastre, antiguo doctorando de un grupo de trabajo de la Universidad de Berekeley formó la compañía Moteiv. Ha desarrollado la plataforma Tmote Sky y Tmote Invent.

SHOCKFISH: Empresa suiza que desarrolla TinyNode.

	Micaz	**Mica2**	**Mica2dot**	**Tmote**	**TinyNode**
Distribuido por	Crossbow			Moteiv	Shockfish
Frecuencia reloj	7.37 MHz		4 MHz	8MHz	8MHz
RAM	4KB			10K bytes	10K bytes
Batería	2 pilas AA		Coin cell	2 pilas AA	Solar
Microcontrolador	Atmel Atmega 128L			Texas Instruments MSP430 microcontroller	

Fig1. Tabla comparativa de motas

Protocolos de MAC y de enrutamiento

Un protocolo de MAC tiene que servir de base para protocolos de más alto nivel, en las redes de sensores, por encima tendríamos el protocolo de enrutamiento, que usará las funciones implementadas en la MAC para enviar y recibir paquetes, sincronizar sus operaciones, etc.

Las características esenciales que debe cubrir un protocolo MAC son:

- La flexibilidad, porque el entorno inalámbrico es totalmente cambiante debido a interferencias en el aire de otras ondas, propiedades y formas de los materiales del entorno, y un largo etcétera. Además, los nodos pueden fallar en cualquier momento, teniendo que buscar nuevos caminos, reconfigurando la red y recalibrando los parámetros. El tráfico puede incrementarse, ya que la información requerida también puede crecer.

- Eficiencia, un protocolo de MAC debe ser eficiente para poder trabajar en tiempo real, debe ser fiable y robusto ante las interferencias. Tolerante a los ruidos. Además debe estar profundamente integrado con el medio donde va a trabajar, a su vez que debe ser un software barato y con funciones que cubran las necesidades más amplias del mercado.

Estos protocolos afectan directamente a la disipación de la energía, ya que son la capa más próxima al nivel físico, también determinará, en parte, el coste del sistema. Así como serán clave a la hora de especificar la latencia y el nivel de seguridad del sistema.

En las redes de sensores estos protocolos determinan los canales de radio a utilizar, implementan las transmisiones y recepciones a bajo nivel, además de controlar los errores.

Las funciones de un protocolo de MAC son controlar el acceso al medio compartido, que en este caso será un canal de radio (a través del aire). El protocolo debe evitar las interferencias entre transmisiones, mitigando el efecto de las colisiones, mediante retransmisiones…
Tenemos varias aproximaciones:

- Basadas en contención, sin coordinación.
- Basadas en planificación, con un nodo central o punto de acceso, encargado de sincronizar al resto.
- División en frecuencia y división en código, también son dos aproximaciones pero no se utilizan prácticamente en las redes inalámbricas de sensores.

Los protocolos de control de acceso al medio (MAC) han sido estudiados durante décadas, los diseños varían mucho según el objetivo de la aplicación. Pueden ser clasificados en categorías basadas en diferentes principios; algunos son centralizados, con una estación central como líder del grupo haciendo el control de acceso, otras son distribuidas. Algunas usan un único canal, otras varios. Algunas usan diferentes versiones de acceso aleatorio; otras usan reserva de canal y planificación. Los protocolos están optimizados para diferentes causas: energía, retardo, tasa de transferencia, equidad, calidad de servicio (QoS) o soportar múltiples servicios.

Cada tipo de red necesitará un protocolo diferente. Por ejemplo, las redes donde los eventos se producen de forma periódica necesitan protocolos que usen reserva y planificación del tiempo. Tendrán una mejor utilización del canal, y tendrán un mayor tiempo de vida. Por el contrario, para las redes de sensores con eventos asíncronos la MAC ha de ser distribuida y optimizada para la energía. Un método distribuido usando múltiples canales y acceso aleatorio sería lo más indicado para estas redes, se evitaría el tener un único punto de fallo, múltiples canales reducen las colisiones y retransmisiones, además del retraso, además incrementa la tasa de transferencia. El acceso aleatorio evita al nodo conocer la red, en este caso no es necesaria la sincronización.

Los nodos no tienen un conocimiento de la topología de la red, deben descubrirla. La idea básica es que cuando un nuevo nodo, al aparecer en una red, anuncia su presencia y escucha los anuncios broadcast de sus vecinos. El nodo se informa acerca de los nuevos nodos a su alcance y de la manera de enrutarse a través de ellos, a su vez, puede anunciar al resto de nodos que pueden ser accedidos desde él. Transcurrido un tiempo, cada nodo sabrá que nodos tiene alrededor y una o más

formas de alcanzarlos.

Los algoritmos de enrutamiento en redes de sensores inalámbricas tienen que cumplir las siguientes normas:

- Mantener una tabla de enrutamiento razonablemente pequeña
- Elegir la mejor ruta para un destino dado (ya sea el más rápido, confiable, de mejor capacidad o la ruta de menos coste)
- Mantener la tabla regularmente para actualizar la caída de nodos, su cambio de posición o su aparición -requerir una pequeña cantidad de mensajes y tiempo para converger

Modelos de enrutamiento

Modelo One-Hop:

Este es el modelo más simple y representa la comunicación directa. Todos los nodos en la red transmiten a la estación base. Es un modelo caro en términos de consumo energético, así como inviable porque los nodos tienen un rango de transmisión limitado. Sus transmisiones no pueden siempre alcanzar la estación base, tienen una distancia máxima de radio, por ello la comunicación directa no es una buena solución para las redes inalámbricas.

Modelo Multi-Hop:

En este modelo, un nodo transmite a la estación base reenviando sus datos a uno de sus vecinos, el cual está más próximo a la estación base, a la vez que este enviará a otro nodo más próximo hasta que llegue a la mota base. Entonces la información viaja de la fuente al destino salto a salto desde un nodo a otro hasta que llega al destino. En vista de las limitaciones de los sensores, es una aproximación viable. Un gran número de protocolos utilizan este modelo, entre ellos todos los MultiHop de Tmote Sky y Telos: MultiHop LQI, MintRoute, Router, etc.

Modelo esquemático basado en clústeres:

Algunos otros protocolos usan técnicas de optimización para mejorar la eficacia del modelo anterior. Una de ellas es la agregación de datos usada en todos los protocolos de enrutamiento basados en clústeres. Una aproximación esquemática rompe la red en capas de clústeres. Los nodos se agruparán en clústeres con una cabeza, la responsable de enlutar desde ese clúster a las cabezas de otros clústeres o la estación base. Los datos viajan desde un clúster de capa inferior a uno de capa superior. Aunque, salta de uno a otro, lo está haciendo de una capa a otra, por lo que cubre mayores distancias. Esto hace que, además, los datos se transfieran más rápido a la estación base. Teóricamente, la latencia en este modelo es mucho menor que en la de MultiHop. El crear clústeres provee una capacidad inherente de optimización en las cabezas de clúster. Por tanto, este modelo será mejor que los anteriores para redes con gran cantidad de nodos en un espacio amplio (del orden de miles de sensores y cientos de metros de distancia).

Aplicaciones de las WSN

- Monitorización de un hábitat (para determinar la población y comportamiento de animales y plantas)
- Monitorización del medio ambiente, observación del suelo o agua
- El mantenimiento de ciertas condiciones físicas (temperatura, luz)
- Control de parámetros en la agricultura
- Detección de incendios, terremotos o inundaciones
- Sensorización de edificios “inteligentes”
- Control de tráfico
- Asistencia militar o civil
- Control de inventario
- Control médico
- Detección acústica
- Cadenas de montaje, etc

2.2 Estándares de comunicación inalámbricos

Cuando nació la era de la computación era difícil concebir la idea de la transmisión de datos a través de ondas de radio. En la actualidad son diversos los estándares que se utilizan, incluso, algunas veces, crean confusiones en su aplicación.

La aparición de los actuales estándares de transmisión de datos y voz se da en forma paralela a sus respectivas industrias y cuando llegan a un punto de encuentro surgen infinitas aplicaciones.

La maduración de las diferentes tecnologías ha sido tan exponencial que en ocasiones la transmisión de los datos invade el terreno de las comunicaciones por voz, sin embargo, cada una de los diferentes estándares tiene alcances y limitaciones.

2.2.1 Zigbee

ZigBee es una alianza, sin ánimo de lucro, de 25 empresas como Invensys, Mitsubishi, Philips y Motorola, con el objetivo del desarrollo de una tecnología inalámbrica de bajo coste

Algunas de las características de ZigBee son:

• ZigBee opera en las bandas libres ISM (Industrial, Scientific & Medical) de 2.4GHz, 868 MHz (Europa) y 915 MHz (Estados Unidos).

• Tiene una velocidad de transmisión de 250 Kbps y un rango de cobertura de 10 a 75 metros.

• A pesar de coexistir en la misma frecuencia con otro tipo de redes como WiFi o Bluetooth su desempeño no se ve afectado, esto se debe a su baja tasa de transmisión y, a características propias del estándar IEEE 802.15.4.

• Capacidad de operar en redes de gran densidad, esta característica ayuda a aumentar la confiabilidad de la comunicación, ya que entre más nodos existan dentro de una red, mayor número de rutas alternativas existirán para garantizar que un paquete llegue a su destino.

• Cada red ZigBee tiene un identificador de red único, lo que permita que coexistan varias redes en un mismo canal de comunicación sin ningún problema.

• Teóricamente pueden existir hasta 16 000 redes diferentes en un mismo canal y cada red puede estar constituida por hasta 65 000 nodos, obviamente estos límites se ven truncados por algunas restricciones físicas (memoria disponible, ancho de banda, etc.).

• Es un protocolo de comunicación multi-salto, es decir, que se puede establecer comunicación entre dos nodos aún cuando estos se encuentren fuera del rango de transmisión, siempre y cuando existan otros nodos intermedios que los interconecten, de esta manera, se incrementa el área de cobertura de la red.

• Su topología de malla (MESH) permite a la red auto recuperarse de problemas e la comunicación aumentando su confiabilidad.

Tipos de dispositivos

Se definen tres tipos diferentes de dispositivos ZigBee según su papel en la red:

• Coordinador ZigBee (ZigBee coordinator, ZC). Puede actuar como director de una red en árbol así como servir de enlace a otras redes. Existe exactamente un coordinador por cada red, que es el nodo que la comienza en principio.

• Router ZigBee (ZR). Además de ofrecer un nivel de aplicación para la ejecución de código programado por el usuario, puede actuar como router interconectando dispositivos separados en la topología de la red.

• Dispositivo final (ZigBee end device, ZED). Posee la funcionalidad necesaria para comunicarse con su nodo padre (el coordinador o un router).

En base a su funcionalidad puede plantearse una segunda clasificación:

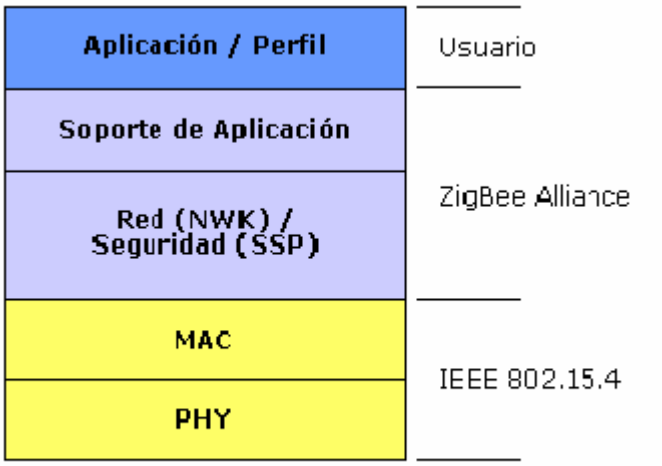
• Dispositivo de funcionalidad completa (FFD): es capaz de recibir mensajes en formato del estándar 802.15.4, puede funcionar como coordinador o router o puede ser usado en dispositivos de red que actúen de interface con los usuarios.

• Dispositivo de funcionalidad reducida (RFD): tiene capacidad y funcionalidad limitadas (especificada en el estándar) con el objetivo de conseguir un bajo coste y una gran simplicidad. Básicamente, son los sensores/actuadores de la red.

Arquitectura

Zigbee es el nombre de la especificación de un conjunto de protocolos de alto nivel de comunicación inalámbrica para su utilización de bajo consumo. Esta basado en el estándar IEEE 802.15.4 de redes inalámbricas de área personal (wíreles personal area network, WPAN). Su objetivo son las aplicaciones que requieren comunicaciones seguras con baja tasa de envío de datos y maximización de la vida útil de sus baterías.

En la Figura se muestran las diferentes capas que conforman la pila de protocolos para ZigBee.



• La capa de más bajo nivel es la capa física (PHY), que en conjunto con la capa de acceso al medio (MAC), brindan los servicios de transmisión de datos por el aire, punto a punto. Estas dos capas esta descritas en el estándar IEEE 802.15.4–2003. El estándar trabaja sobre las bandas ISMde uso no regulado, dónde se define hasta 16 canales en el rango de 2.4GHz, cada una de ellas con un ancho de banda de 5 MHz. Se utilizan radios con un espectro de dispersión de secuencia directa, lográndose tasas de transmisión en el aire de hasta 250 Kbps en rangos que oscilan entre los 10 y 75 m, los cuales dependen bastante del entorno.

• La capa de red (NWK) tiene como objetivo principal permitir el correcto uso del subnivel MAC y ofrecer una interfaz adecuada para su uso por parte de la capa de aplicación. En esta capa se brindan los métodos necesarios para: iniciar la red, unirse a la red, enrutar paquetes dirigidos a otros nodos en la red, proporcionar los medios para garantizar la entrega del paquete al destinatario final, filtrar paquetes recibidos, cifrarlos y autentificarlos. Es en esta capa en donde se implementan las distintas topologías de red que ZigBee soporta (árbol, estrella y mesh network).

• La siguiente capa es la de soporte a la aplicación que es el responsable de mantener el rol que el nodo juega en la red, filtrar paquetes a nivel de aplicación, mantener la relación de grupos y dispositivos con los que la aplicación interactúa y simplificar el envío de datos a los diferentes nodos de la red. En el nivel conceptual más alto se

encuentra la capa de aplicación que no es otra cosa que la aplicación misma y de la que se encargan los fabricantes.

IEEE 802.15.4

Es un estándar que define el nivel físico y el control de acceso al medio de redes inalámbricas de área personal con tasas bajas de transmisión de datos (*low-rate wireless personal area network*, LR-WPAN). La actual revisión del estándar se aprobó en 2006. El grupo de trabajo IEEE 802.15 es el responsable de su desarrollo.

También es la base sobre la que se define la especificación de ZigBee, cuyo propósito es ofrecer una solución completa para este tipo de redes construyendo los niveles superiores de la pila de protocolos que el estándar no cubre.

El propósito del estándar es definir los niveles de red básicos para dar servicio a un tipo específico de red inalámbrica de área personal (WPAN) centrada en la habilitación de comunicación entre dispositivos ubicuos con bajo coste y velocidad (en contraste con esfuerzos más orientados directamente a los usuarios medios, como Wi-Fi). Se enfatiza el bajo coste de comunicación con nodos cercanos y sin infraestructura o con muy poca, para favorecer aún más el bajo consumo.

En su forma básica se concibe un área de comunicación de 10 metros con una tasa de transferencia de 250 kbps. Se pueden realizar compromisos que favorezcan aproximaciones más radicales a los sistemas empotrados con requerimientos de consumo aún menores. Para ello se definen no uno, sino varios niveles físicos. Se definieron inicialmente tasas alternativas de 20 y 40 kbps; la versión actual añade una tasa adicional de 100 kbps. Se pueden lograr tasas aún menores con la consiguiente reducción de consumo de energía. Como se ha indicado, la característica fundamental de 802.15.4 entre las WPAN's es la obtención de costes de fabricación excepcionalmente bajos por medio de la sencillez tecnológica, sin perjuicio de la generalidad o la adaptabilidad.

Entre los aspectos más importantes se encuentra la adecuación de su uso para tiempo real por medio de slots de tiempo garantizados, evitación de colisiones por CSMA/CA y soporte integrado a las comunicaciones seguras. También se incluyen funciones de control del consumo de energía como calidad del enlace y detección de energía.

2.2.2 Wi-Fi

Es un estándar de protocolo de comunicaciones del IEEE que define el uso de los dos niveles inferiores de la arquitectura OSI (capas física y de enlace de datos), especificando sus normas de funcionamiento en una WLAN. En general, los protocolos de la rama 802.x definen la tecnología de redes de área local.

La familia 802.11 actualmente incluye seis técnicas de transmisión por modulación, todas las cuales utilizan los mismos protocolos. El estándar original de este protocolo data de 1997, era el IEEE 802.11, tenía velocidades de 1 hasta 2 Mbps y trabajaba en la banda de frecuencia de 2,4 GHz.

El término IEEE 802.11 se utiliza también para referirse a este protocolo al que ahora se conoce como "802.11legacy." La siguiente modificación apareció en 1999 y es designada como IEEE 802.11b, esta especificación tenía velocidades de 5 hasta 11 Mbps, también trabajaba en la frecuencia de 2,4 GHz. También se realizó una especificación sobre una frecuencia de 5 Ghz que alcanzaba los 54 Mbps, era la 802.11a y resultaba incompatible con los productos de la b y por motivos técnicos casi no se desarrollaron productos. Posteriormente se incorporó un estándar a esa velocidad y compatible con el b que recibiría el nombre de 802.11g. La versión final del estándar se publicó en Junio de 2007 y recoge las modificaciones más importantes sobre la definición original; incluye: 802.11a, b, d, e, g, h, i, j.

En la actualidad la mayoría de productos son de la especificación b y de la g. El siguiente paso se dará con la norma 802.11n que sube el límite teórico hasta los 600 Mbps. Actualmente ya existen varios productos que cumplen un primer borrador del estándar N con un máximo de 300 Mbps (80-100 estables).

La seguridad forma parte del protocolo desde el principio y fue mejorada en la revisión 802.11i. Otros estándares de esta familia (c–f, h–j, n) son mejoras de servicio y extensiones o correcciones a especificaciones anteriores. El primer estándar de esta familia que tuvo una amplia aceptación fue el 802.11b. En 2005, la mayoría de los productos que se comercializan siguen el estándar 802.11g con compatibilidad hacia el 802.11b.

Los estándares 802.11b y 802.11g utilizan bandas de 2,4 Ghz que no necesitan de permisos para su uso. El estándar 802.11a utiliza la banda de 5 GHz. El estándar 802.11n hará uso de ambas bandas, 2,4 GHz y 5 GHz. Las redes que trabajan bajo los estándares 802.11b y 802.11g pueden sufrir interferencias por parte de hornos microondas, teléfonos inalámbricos y otros equipos que utilicen la misma banda de 2,4 Ghz.

802.11a

La especificación IEEE 802.11a hace uso de la banda a los 5 GHz. Aquí no se emplea un esquema de espectro expandido, sino multiplexado por división de frecuencia ortogonal.

Las velocidades de datos posibles son 6, 9, 12, 18, 24, 36, 48 y 54 Mbps.
Cuenta con un número adicional de canales sin solapamiento que aumenta el rendimiento al tiempo y la facilidad de ampliación.

No es compatible con 802.11b ni con 802.11g, pero puede coexistir con estos estándares sin repercutir en el rendimiento.

802.11b

Tiene una velocidad máxima de datos de hasta 11 Mbps por canal, mayor que la mayoría de las conexiones cableadas de banda ancha.

Ofrece una penetración mejor a través de muros y otros obstáculos, y mayor alcance en la oficina, el hogar y otros entornos cerrados. Es con mucha diferencia el estándar inalámbrico más ampliamente usado

802.11g

Es una extensión de IEE 802.11b a mayor velocidad. Este esquema combina toda una gama de técnicas de codificación del medio físico utilizadas en 802.11a y 802.11b para proporcionar servicio a diversas velocidades de datos.

Compatible con 802.11b, constituye una buena opción para empresas cuyos usuarios utilicen portátiles con el estándar 802.11b y que necesiten más velocidad para futuras aplicaciones.

2.2.3 Bluetooth

Bluetooth es un sistema de comunicaciones de corto alcance, por medio de radiofrecuencia, cuyo objetivo es eliminar los cables en las conexiones entre dispositivos electrónicos, tanto portátiles como fijos, manteniendo altos niveles de seguridad. Las características de esta tecnología son su fiabilidad, bajo consumo y mínimo costo. La especificación Bluetooth establece una organización uniforme para que un amplio abanico de dispositivos pueda conectarse y comunicarse entre sí. Los objetivos a conseguir con esta norma son:

- Facilitar las comunicaciones entre equipos móviles y fijos.
- Eliminar cables y conectores entre estos.
- Ofrecer la posibilidad de crear pequeñas redes inalámbricas y facilitar la sincronización de datos entre los equipos personales.

El alcance que logran tener estos dispositivos es de 10 metros para ahorrar energía ya que generalmente estos dispositivos utilizan mayoritariamente baterías. Sin embargo, se puede llegar a un alcance de hasta 100 metros (similar a Wi-Fi), pero aumentando el consumo energético considerablemente. Para mejorar la comunicación no debe interponerse nada físico (como una pared).

A diferencia de los infrarrojos, no se tiene que colocar los dispositivos directamente frente a frente, tampoco deben estar en la misma habitación. Bluetooth puede activar conexiones de forma automática con los dispositivos asociados, de modo que ni siquiera hay que pensar en ello. Y no se tiene que pagar por una conexión Bluetooth, no importa la cantidad de datos que se transmita.

Las tecnologías existentes son:

- Bluetooth v.1.1
- Bluetooth v.1.2
- Bluetooth v.2.0

La versión 1.2, a diferencia de la 1.1, provee una solución inalámbrica complementaria para coexistir Bluetooth y Wi-Fi en el espectro de los 2.4 GHz, sin interferencia entre ellos. Y llegan a transmitir a 1Mbps.

La versión 1.2 ejecuta una más eficiente transmisión y más seguro encriptamiento (herramienta de seguridad). También la versión 1.2 disminuye el ruido ambiental. Además, provee una configuración más rápida de la comunicación con los otros dispositivos dentro del rango del alcance, como pueden ser PDAs, HIDs (Human

Interface Devices), ordenadores portátiles, ordenadores de sobremesa, Headsets, impresoras, celulares, y hasta instrumentales médicos.

La versión 2.0, creada para ser una especificación separada, permite mejorar las velocidades de transmisión hasta 3Mbps.También intenta solucionar algunas equivocaciones de la especificación 1.2.

La tecnología inalámbrica Bluetooth está orientada a aplicaciones de voz y datos. Funciona en la banda de frecuencia de 2.4 GHz, que no precisa de ninguna licencia. Tiene un radio de acción de 10 o 100 metros dependiendo de la clase del dispositivo Bluetooth. La máxima velocidad de transmisión es de 3 Mbps.

Los objetos sólidos no suponen ningún obstáculo para la tecnología inalámbrica Bluetooth. Tampoco es necesario que los dispositivos estén situados en la misma línea de visión, es decir, orientados uno frente a otro, ya que se transmite en todas las direcciones.

2.2.4 Comparativa entre estándares.

En la siguiente tabla se muestra una comparativa entre los distintos estándares analizados anteriormente:

ESTÁNDAR	WI-FI 802.11g	WI-FI 802.11b	BLUETOOTH V 1.1	ZIGBEE 802.15.4
Aplicación principal	WLAN	WLAN	WPAN	Control y monitorización
Memoria necesaria	1MB+	1MB+	250KB+	4KB-32KB
Vida Batería (días)	0.5 - 5	0.5 - 5	1 - 7	100 - 1000
Velocidad (Kbps)	54 Mbps	11 Mbps	1Mps	20 – 250 Kbps
Cobertura (metros)	100	100	1 - 10	1 – 100
Parámetros más importantes	Velocidad y flexibilidad	Velocidad y Flexibilidad	Costes y perfiles de aplicación	Fiabilidad, bajo consumo y muy bajo coste

2.3 Red ZigBee desarrollada en el puerto de Génova

La red de sensores inalámbrica Zigbee a monitorizar fue realizada por el Departamento de Ingeniería Biofísica y Electrónica (DIBE) de la Universidad de Génova. Está instalada sobre un sistema de contadores de consumo eléctrico ubicado en el puerto antiguo de Génova.

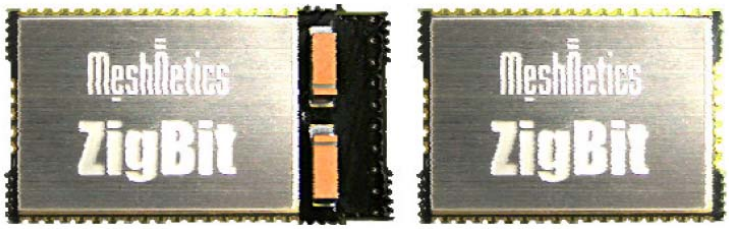
En la primera fase del proyecto se estudiaron varias soluciones y se realizó un estudio de mercado y de características, destacando los siguientes módulos:

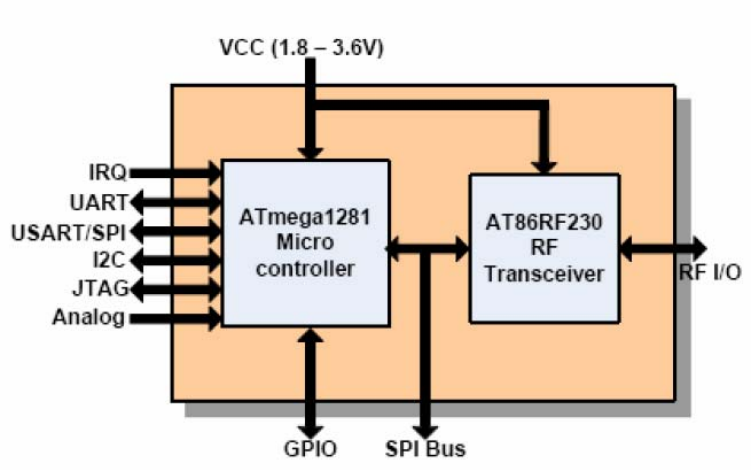
- ETRX2 (Telegesis).
- Zigbit (Meshnetics).
- XBee Serie 2 (MaxStream).

Finalmente la solución que presentaba mayores ventajas era la de Zigbit de Meshnectics. Las principales características del módulo se presentan a continuación.

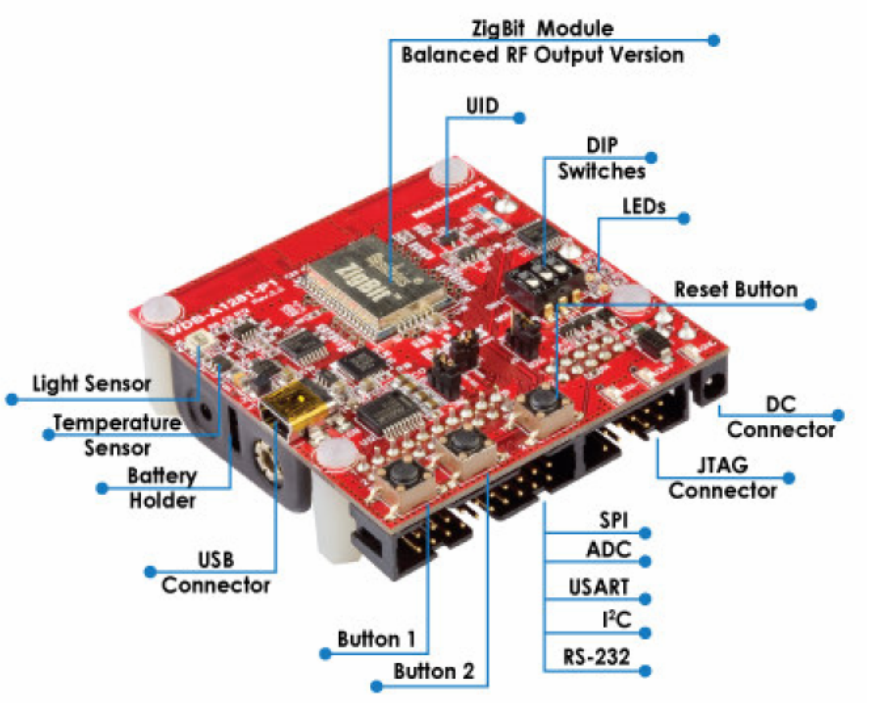
HARDWARE	SOFTWARE
Microcontrolador y Módulo RF: se utiliza una Atmega1281 Atmel (4 MHz, 128 kB FLASH, 8 kB RAM, 4 kB EEPROM) pertenece a la familia AVR 8-bit RISC. El módulo RF es un AT86RF230 Atmel, esto es un transmisor de baja potencia a 2,4 GHz desarrollado para Atmel por ZigBee/IEEE802.15.4. **Interfase externa:** - 10 GPIO, 2 líneas IRQ. - 4 líneas ADC. - UART, I2C, SPI. **Opciones para la salida RF:** - Antena cerámica integrada (dipolo simétrico). - Conector coaxial. - Pad de 50 Ω (por ejemplo para antena PCB). **Potencia consumida:** < 1μA en "deep sleep mode" (el wake-up puede venir por la interrupciones externas o desde el timer. Todavía no se ha desarrollado el soporte para las interrupciones con comandos AT). **Dimensiones:** 24 x 13,5 mm (con antena cerámica integrada)	**Stack ZigBee:** eZeeNet stack desarrollado para Meshnetics en colaboración con Atmel. Es un stack que se programa con API. **Firmware preinstalado:** eZeeNet stack; es necesario cargar en memoria un programa para gestionar el dispositivo. El ambiente de desarrollo es AVR Studio de Atmel y se puede descargar WinAVR, el cual abastece al compilador de GCC C free, para poder programar el microcontrolador Atmel presente en el chip. En alternativa a la programación con las API se puede cargar el firmware SerialNet, que permite al módulo una interfase con comandos AT. En estos momentos esta solución esta limitada debido a que no tiene soporte para: - USART/I2C. - IRQ externa. - Algunas funcionalidades del estándar ZigBee. - Actualizar el firmware vía wireless a todos los otros nodos de la red.

En las siguientes imágenes se puede ver una foto del chip así como el diagrama de bloques que componen el dispositivo.





Para el aprendizaje con el módulo fue utilizado el sistema de desarrollo Meshbean 2, del cual se presenta una imagen a continuación.

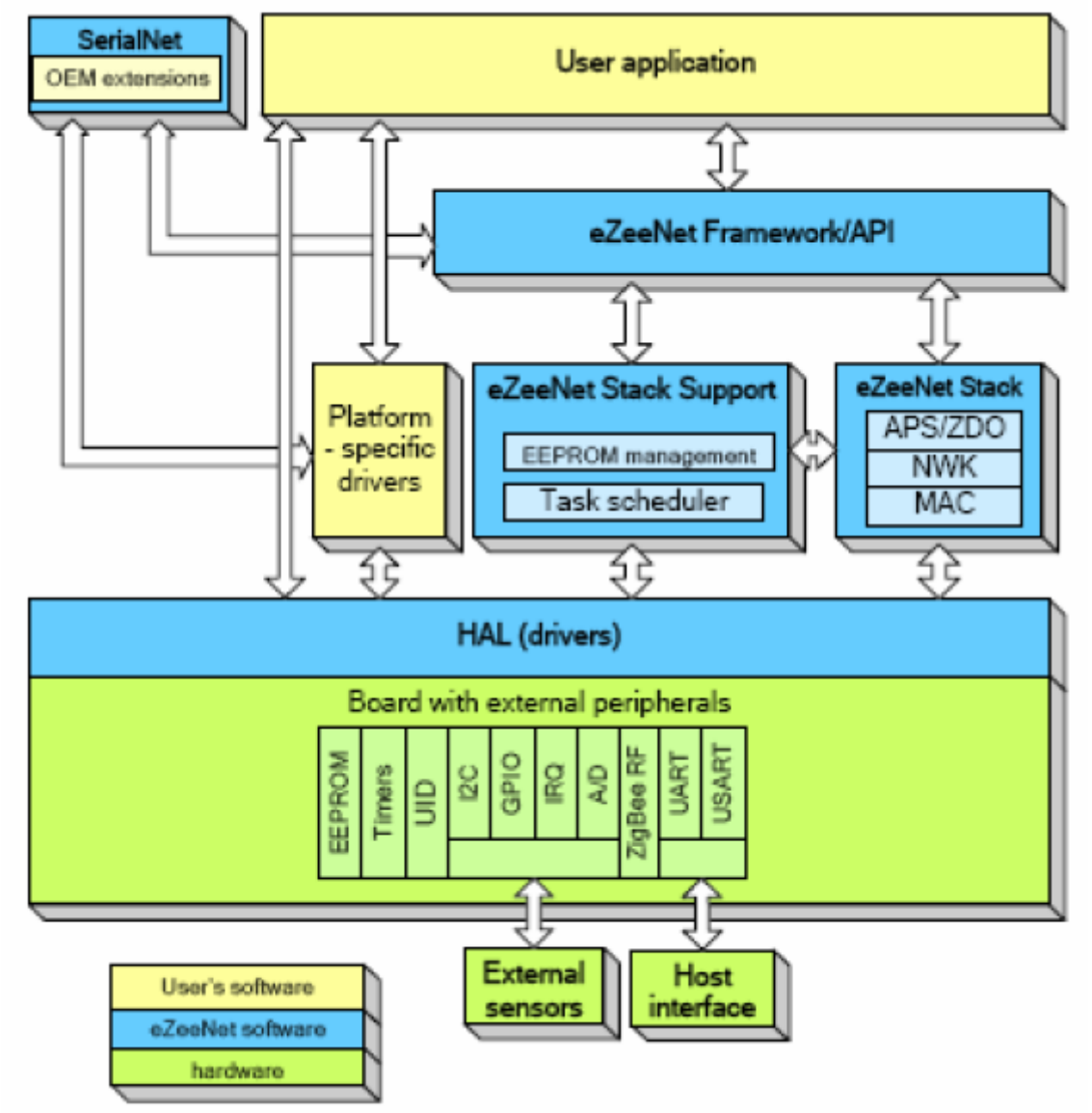


EZeeNet Stack

El stack software proporcionado por Meshnectics compatible con el estándar ZigBee 2006 se llama eZeeNet.

El software utiliza el sistema operativo TinyOS en particular el scheduler presente en su interior, para gestionar los task de bajo nivel, tales como las interrupciones por parte del AT86R230, distribuyendo al cliente las funciones de base para gestionar la comunicación ZigBee.

Estas funcionalidades son llamadas en funciones C, el sistema operativo, una vez ejecutado el comando solicitado al cliente, realizará otra vez las funciones de Callback, en modo tal que el mismo cliente sea informado del éxito del comando pedido.

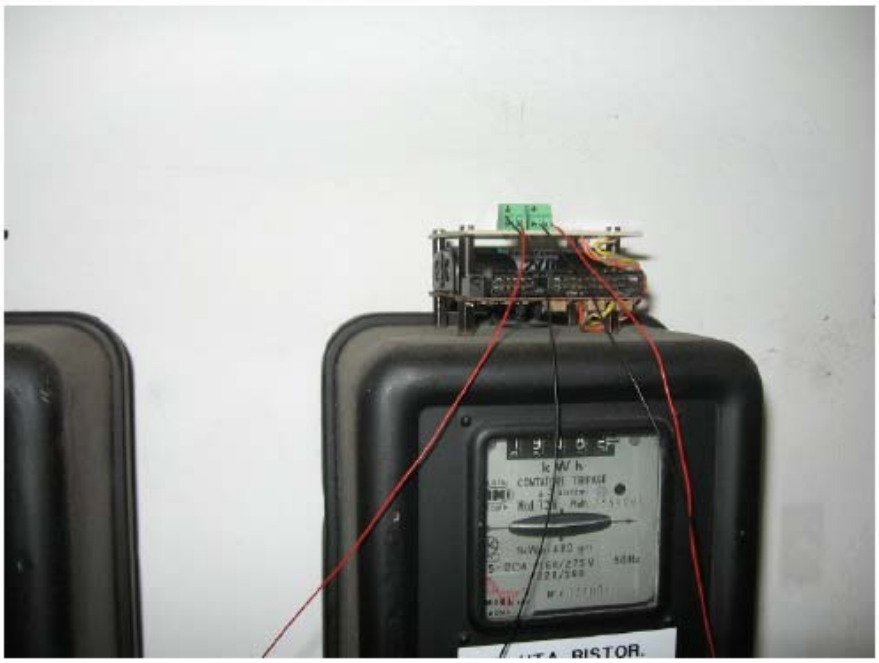


Mariano Gallardo Magaña
Ingeniería en Automática y Electrónica Industrial

PRIMER PROTOTIPO

A continuación se muestra un primer prototipo de la funcionalidad de los módulos wireless para las lecturas de los consumos eléctricos de los contadores del Puerto Antiguo de Génova utilizando las tarjetas Meshbean 2.

Específicamente se realizo una simple tarjeta de adquisición para conseguir la lectura de dos contadores eléctricos.



Los contadores presentes en el área son los Vermer modelo Energy-400, que poseen una salida opto acoplada que proporciona un impulso por cada KWh consumido por el cliente conectado al contador eléctrico.

Mariano Gallardo Magaña
Ingeniería en Automática y Electrónica Industrial

Para leer los impulsos proporcionados por el contador han sido utilizadas directamente las líneas de interrupción del Zigbit en las pull-up internas al microcontrolador, conectados al colector del transistor de salida del opto acoplador. De esta forma siempre se puede mantener el modo sleep el nodo sensor y hacerlo despertar directamente cuando se recibe un impulso de contador, una vez recibido el impulso se ejecuta la rutina de interrupción.

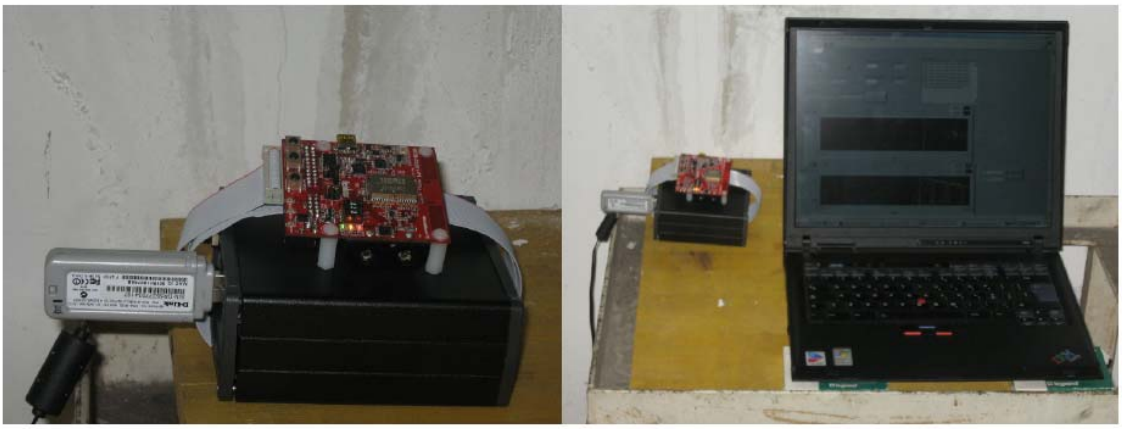
La estrategia utilizada para desarrollar el firmware ha sido la siguiente, el nodo sensor esta en modo sleep la mayor parte del tiempo y los impulsos de los contadores son leídos a través de las rutinas de interrupción del microcontrolador presente dentro de ZigBit.

Los valores de los impulsos contados en el periodo de tiempo transcurrido son salvados en una localización de la memoria no volátil EEPROM, la cual es actualizada cada vez que se recibe un impulso.

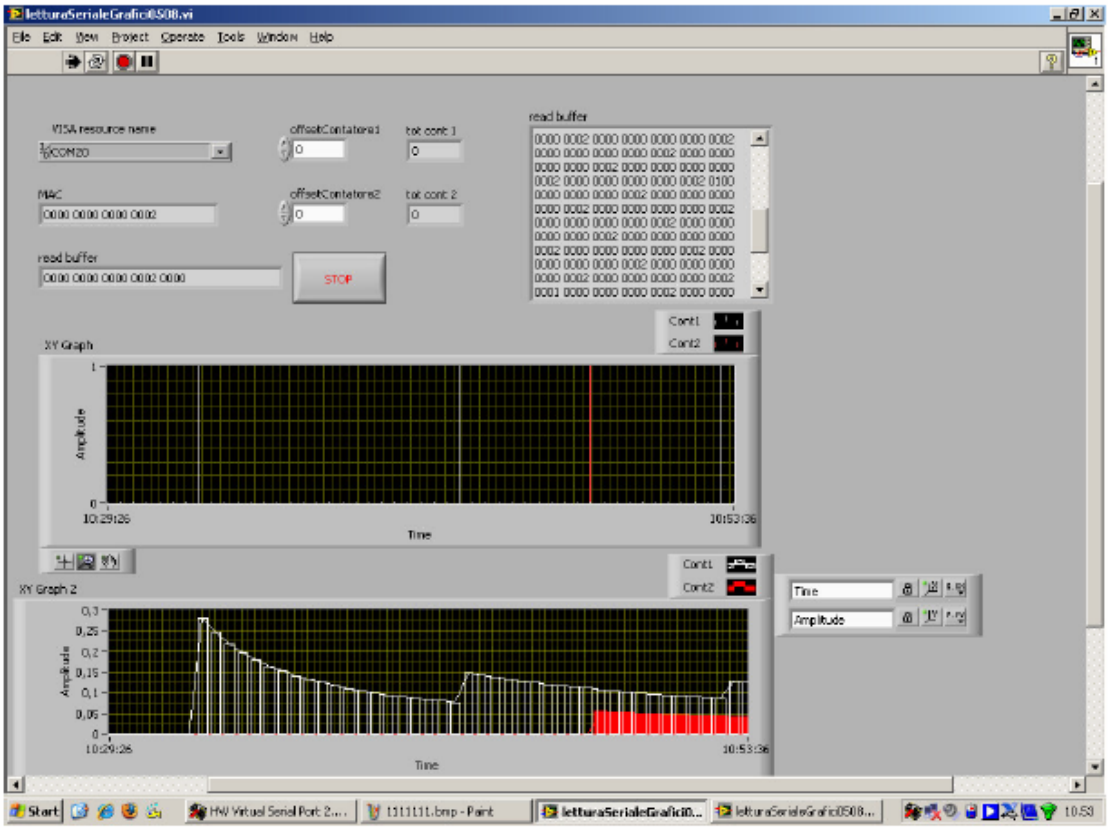
Periódicamente después de un intervalo de tiempo fijado, cada nodo sensor envía el valor del número de impulsos que ha almacenado vía radio al coordinador de la red ZigBee.

Si la recepción de la información es correcta por parte del coordinador el nodo sensor recibe un acknowledge (ACK), esto hará que el nodo pueda liberar dicha memoria EEPROM para almacenar nuevos impulsos, en caso contrario es decir si sucede cualquier error en el envío de la información, el nodo sensor seguirá almacenando el valor de los impulsos.

El coordinador es conectado a la FoxBoard y como comunica con ésta vía serie. Para realizar la interfase con un PC, la FoxVoard es dotada de un dongle WiFi USB. La comunicación WiFi es utilizada para permitir una conexión sin cables. El coordinador remite los datos recibidos desde los nodos sensores a través de su interfase serie y la FoxBoard los manda al PC a través de WiFi.

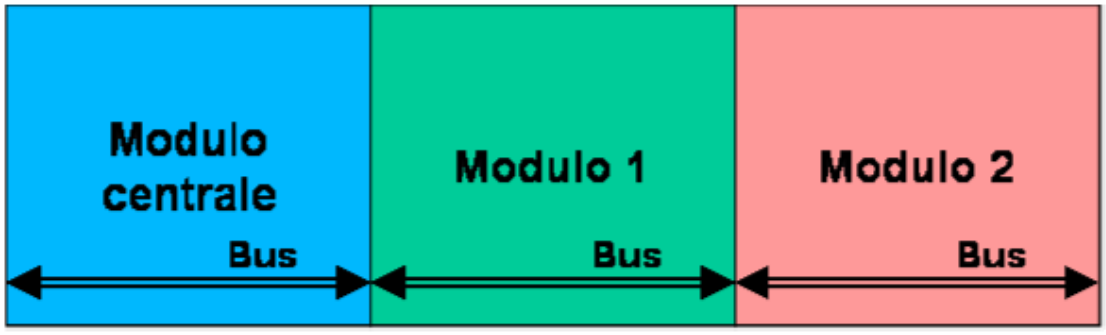


Para facilitar la visualización de los resultados de los nodos sensores, en el PC se ha desarrollado una aplicación LabView, así se consigue visualizar de forma gráfica el consumo de los contadores.



Arquitectura general del nodo sensor

El nodo sensor ha sido proyectado de forma que se tenga una estructura modular, este quiere decir, tener un módulo de transmisión wireless compatible con el estándar 802.15.4/ZigBee común en todas las aplicaciones y diferentes módulos de adquisición según los requisitos de monitorización.



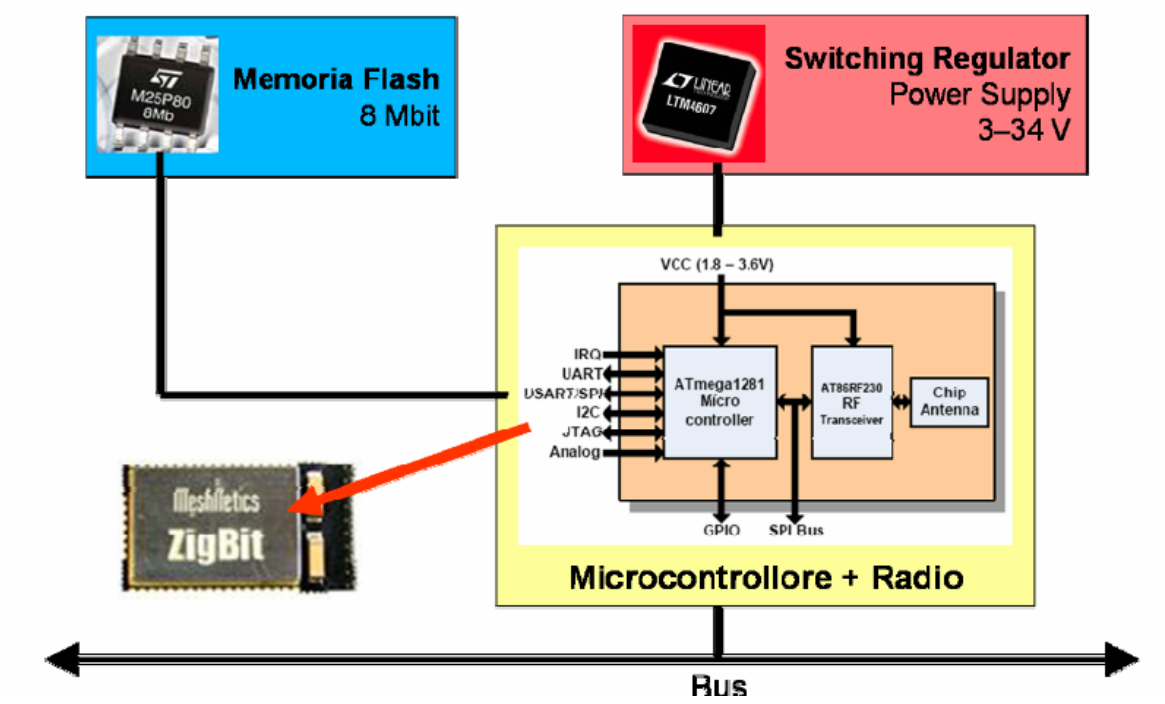
De esta forma es posible utilizar solo el hardware necesario para una determinada aplicación y se reducen los consumos, debido a que no se añaden componentes que puede que no sean utilizados.

El sistema entera ha sido proyectado para ser montado sobre una guía DIN, como típicamente son montados los contadores eléctricos y otros dispositivos industriales.

Módulo central del nodo sensor

El módulo de transmisión o modulo central del nodo sensor contiene en su interior:

- ZigBit con antena tipo dipolo integrada.
- Memoria Flash M25P80 de 8Mbit (ST Microelectronic).
- Un circuito de alimentación capaz de recibir a la entrada una tensión comprendida en el rango de 5 a 34 V y de proporcionar 3,3 V a los otros módulos de la tarjeta, incluido ZigBit.
- Divisor de tensión para medir el estado de la batería conectada al módulo.
- Un dip-swith para consentir al instalador entre distintas configuraciones, así como seleccionar el tipo de alimentación.

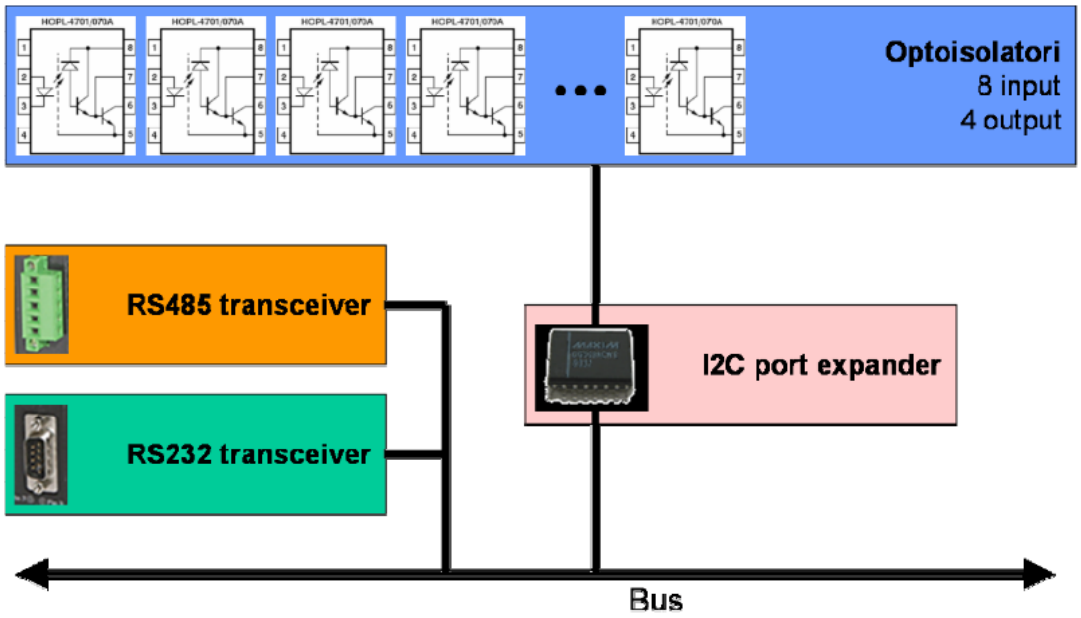


Mariano Gallardo Magaña
Ingeniería en Automática y Electrónica Industrial

Módulo de adquisición RS232 y RS485.

La tarjeta de adquisición desarrollada para la aplicación industrial contiene:

- Un transceiver RS485 (MAX3471) para bajo consumo.
- Un transceiver RS232 (MAX3222) también para bajo consumo, para conseguir una comunicación serie a cuatro hilos (esto quiere decir con RTS, CTS, RX y TX).
- Un puerto de expansión I2C (MAX7324) con 8 entradas y 8 salidas digitales.
- Opto acopladores (HCPL-073L y HCPL-073A) para desacoplar eléctricamente las 8 entradas del puerto de expansión I2C.
- Un dip-switch para activar los opto acopladores y las interfases series RS485 y RS232.



Protocolo de comunicación entre los nodos wíreles del puerto antiguo

Se ha creado un protocolo de comunicación entre los distintos nodos colocados en el puerto antiguo de Génova para realizar las medidas de consumo y transmitir dichas medidas al nodo central.

3 OBJETIVOS DEL PROYECTO

El objetivo principal de este proyecto es desarrollar un software para una aplicación de monitorización sobre una red de sensores inalámbrica.

La red de sensores está formada por el sistema de contadores de consumo eléctrico de las estaciones de San Giobatta, San Lorenzo y San Desiderio, del puerto de la ciudad de Génova (Italia).

La aplicación software, desarrollada en J2ME con el entorno Eclipse, correrá en un teléfono móvil Nokia 6131 NFC. Podremos visualizar en un teléfono móvil el consumo eléctrico diario de las estaciones.

Las mediciones que tomen cada uno de los nodos contadores se enviarán inicialmente hacia una estación base que recogerá y almacenará los resultados. El software desarrollado en este proyecto permitirá al teléfono móvil comunicarse con esta estación base vía Bluetooth y obtener información de los grupos contadores para después monitorizarla.

4 SOLUCIÓN PARA LA MONITORIZACIÓN DE CONTADORES

4.1 Estación Base Sistema Bluetooth

La estación Base ha sido desarrollada en otro proyecto llamado:"Sistema pasarela Bluetooth para red de Sensores Zigbee" por Daniel Gutiérrez Reina, en el título se puede apreciar la función de lo que en este proyecto llamamos estación base ya que el Sistema Bluetooth desarrollado por Daniel Gutiérrez hace de puerta pasarela entre la red de sensores Zigbee y el dispositivo móvil que alberga el sistema de monitorización.

Dentro del diseño del dispositivo para la comunicación Bluetooth se realizó un estudio de mercado de los dispositivos que actualmente proporcionan los fabricantes de módulos IC para telecomunicaciones, buscando las siguientes características

- Bajo consumo, ya que dentro de las redes de sensores este es un requisito básico debido a la necesidad de baterías.
- Bajo costo, fundamental cuando se quiere realizar un desarrollo para su posterior producción y venta.
- Dimensiones, es importante porque el sistema debe resultar lo más pequeño posible.
- Microcontrolador embebido, es importante para poder programar el módulo de forma de que las aplicaciones puedan ser implantadas dentro de él y no tener la necesidad de comunicar el sistema Bluetooth con un controlador externo.
- Antena, existen módulos que tiene la antena integrada y otros que tienen el conector para añadirle una externa.
- Software y tarjeta de desarrollo para poder desarrollar aplicaciones sobre el microcontrolador.
- Medio o largo alcance, dentro de los módulos Bluetooth existen diversas clases, clase 1, 2 y 3. Dependiendo de la clase, la comunicación se puede realizar a más o menos alcance. Para esta aplicación es necesario un módulo de clase 1 ò 2.
- Sensibilidad en el receptor y potencia en el transmisor Bluetooth.
- Memoria disponible para el almacenamiento y manipulación de la información.
- Tasa de transmisión.

- Tensión de alimentación.
- Posibilidad de realizar encriptación de datos.
- Encapsulado compatible con las posibilidades de desarrollo.

Los módulos que mejor cumplían estas características fueron:

-GS – BT2416C2.AT1 (ST – Microelectronic)
-F2M03GLA (FreeMove)
-RN 21 (RovingNetworks)
-WT11 (Bluegiga)
-PAN 1550 (Panasonic)

Entre los que acabo eligiéndose el WT11 de Bluegiga, que contiene todo lo necesario para poder implementar una comunicación Bluetooth, bloque de radio, antena y la pila completa de protocolos. Además viene equipado con el firmware iWRAP, que permite a los usuarios acceder al módulo mediante unos sencillos comandos ASCII. Aunque la característica más importante a la hora de realizar la elección fue el entorno de trabajo que proporciona Bluegiga, proporcionando las herramientas de desarrollo necesarias para realizar el desarrollo del sistema de una forma efectiva y sencilla. También a la hora de realizar código sobre el mismo microprocesador embebido en el módulo, gracias al kit de desarrollo Software Bluelab, que permite realizar código a medida y que será muy útil a la hora de darle inteligencia al sistema.

4.2 Teléfono móvil Nokia 6131 NFC

El teléfono móvil es el soporte físico encargado de albergar y ejecutar el software de monitorización, por lo tanto, debe tener las características de programabilidad, comunicación y de visualización suficientes, además de poder ejecutar aplicaciones desarrolladas libremente por cualquier usuario.

La plataforma S40 de Nokia ofrece dos opciones de desarrollo para crear aplicaciones que puedan ser instaladas en un dispositivo: la plataforma Java, tecnología micro edition (Java ME), con configuración CLDC 1.1 y el perfil de información de dispositivo móvil (MIDP) 2.0 o 2.1.

Además la plataforma Series 40 suministra un juego común de tecnologías y APIs para el desarrollo de aplicaciones y la distribución de contenidos digitales. La plataforma incluye un juego de APIs de Java, tecnología MIDP y CLDC, que junto con los paquetes oportunos de JSRS, provee al dispositivo de más herramientas y aplicaciones para comunicaciones, mensajería, multimedia y capacidad de gráficos. El ejemplo ideal para nuestro proyecto es que existe un interfaz para el desarrollo de aplicaciones JSR-82 que nos facilita librerías, métodos, funciones y clases para desarrollar aplicaciones Java que hagan uso de Bluetooth

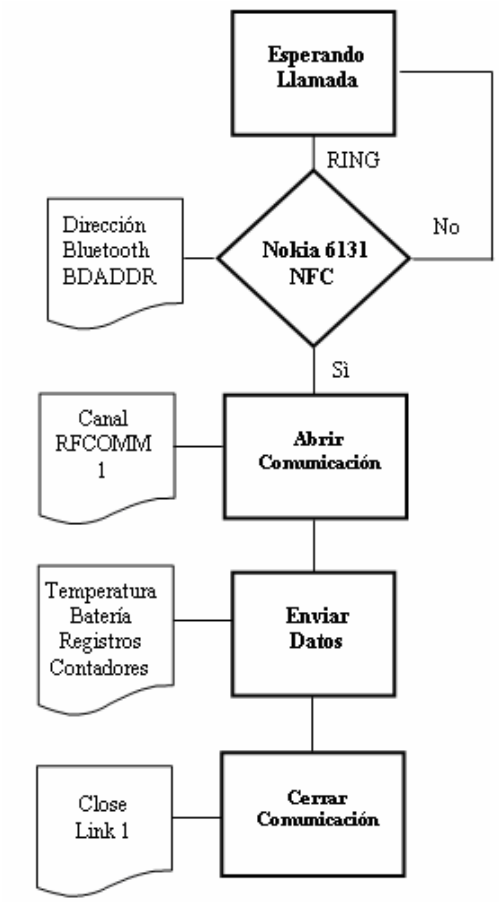
Por lo tanto el móvil Nokia 6131 NFC, que funciona bajo la plataforma Series 40, disponible en el DIBE de la Universidad de Génova, ha sido el teléfono móvil elegido para desarrollar y ejecutar la aplicación de monitorización de este proyecto.

4.3 Comunicación Estación Base Bluetooth-Teléfono móvil

Los datos captados por la red de sensores Zigbee y que reflejen el consumo de los Grupos Contadores se almacenan en la Estación Base Bluetooth. Desde allí, y cuando lo solicite nuestro sistema de monitorización por petición del usuario, serán enviados a través del protocolo de comunicación Bluetooth al teléfono móvil.

Se ha utilizado el perfil serie para realizar la transmisión de información, es un perfil que permite emular un puerto serie utilizando como medio de transmisión la comunicación Bluetooth. Esto quiere decir que la información llegará al receptor, es decir el teléfono móvil, de forma continua. En concreto el teléfono recibirá una cadena de caracteres que contendrá todo la información de interés para el MIDlet de monitorización que corre sobre el teléfono móvil.

El siguiente diagrama de flujo sirve para entender como se realiza los pasos para establecer la comunicación y enviar la información al teléfono móvil.



4.3.1 Establecimiento de la comunicación

El establecimiento de la comunicación se realiza una vez que se ha recibido una llamada por parte del teléfono móvil, mediante software se comprueba que la llamada corresponde al teléfono móvil Nokia 6131 NFC, esta comprobación se realiza comparando la dirección Bluetooth del teléfono, conocida de antemano por la estación base sistema Bluetooth, con la dirección Bluetooth del dispositivo que ha realizado la llamada.

En el caso de que la dirección Bluetooth que se recibe no coincide con la del teléfono se cierra inmediatamente el canal de comunicación que se había abierto, volviendo el sistema al estado inicial de reposo en el que espera una nueva solicitud de comunicación.

4.3.2 Envío de Información

Una vez establecida la comunicación, el sistema Bluetooth envía la trama de caracteres por el canal de comunicación serie abierto, un canal del protocolo RFCOMM que se abre cuando se establece la comunicación.

En la siguiente figura se puede observar en que consiste la trama de información que se envía al teléfono:

Temperatura (4 Bytes)	Batería (6 Bytes)	Registro de los Contadores (n Bytes)

En primer lugar se envían 4 bytes que corresponden con la medida de temperatura que mide el sensor que lleva incorporado el WT-11. Se utilizan cuatro bytes, de los que corresponden 2 bytes al valor numérico de la temperatura con dos cifras y otros dos bytes para enviar los caracteres ºC. Un posible ejemplo para este campo sería 25ºC.

A continuación se mandan 6 bytes, estos corresponden con el % de batería disponible del sistema montado en la estación base en el momento en el que se realiza la solicitud de comunicación. De estos 6, 5 bytes se utilizan para las cifras numéricas del porcentaje, incluyendo dos dígitos del valor entero más el carácter de separación de decimales y el valor decimal (dos dígitos). El sexto byte corresponde al símbolo porcentual. Un posible valor para este campo sería el de 75,20%.

Con la parte final de la trama de caracteres se envían los registros que contienen información acerca del consumo de los contadores. Se incluye la cantidad de registros nuevos disponibles en la estación base para ser enviados.

Cada registro esta separado del siguiente por el carácter guión “-“ y representa el consumo de ¼ de KW/h por el grupo contador indicado, en la fecha señalada, y en el instante de tiempo indicado.

La siguiente figura constituye lo que sería un registro de información

-	1	:	1	4	#	0	3	#	2	0	0	9	:	1	0	*	0	0

- El carácter '-'corresponde a la indicación del comienzo de nuevo registro.
- El siguiente byte en el ejemplo , el nº 1, corresponde al número de Grupo Contador. Existen 3 posibles Grupos Contadores

Grupos Contadores
GRUPO 1 SAN DESIDERIO
GRUPO 2 SAN LORENZO
GRUPO 3 SAN GIOBATTA

- El carácter ':' sirve de separación para los campos de Grupo, fecha e instante.
- La fecha se manda en formado "dd#mm#aaaa", donde el carácter # sirve para diferenciar los distintos subcampos que componen la fecha.
- En último lugar se envía el instante en formato "hh*mm", en este caso se utiliza el carácter '*' para diferenciar los subcampos.

  Nota: La forma y caracteres utilizados por esta trama de envío tiene su explicación en el apartado que contiene la descripción del software desarrollado en este proyecto, apartado llamado : "MIDlet de monitorización de los contadores del puerto de Génova"

4.3.3 Cierre de la comunicación

El cierre de la comunicación lo realiza el sistema Estación Base Bluetooth cuando termina de enviar la trama completa. El sistema cierra automáticamente el canal RFCOMM, que previamente había abierto al recibir la llamada por parte del teléfono.

5 SISTEMA DE MONITORIZACIÓN

En este apartado se presenta y desarrollan todos los dispositivos, entornos de desarrollo y tecnologías involucradas en la creación y desarrollo del software de monitorización objetivo de este proyecto.

5.1 Dispositivo de Telefonía Móvil Nokia 6131 NFC

5.1.1 Especificaciones Técnicas

Frecuencia operativa

- Quad-band GSM/EDGE. Capacidad de cobertura en los 5 continentes (850/900/1800/1900)

Medidas

- Volumen: 75 cc
- Peso: 104 g
- Dimensiones: 92 x 47 x 20 mm

Visualización e interfaz de usuario

- Visualización principal: Pantalla TFT 2.2” QVGA con más de 16.7 millones de colores (240 x 320 píxeles)
- Pantalla externa TFT de 1.36 " (128 x 160 píxeles)
- Interfaz de usuario de Series 40 mejorada

Cámara

- 1.3-megapíxeles con 8x de zoom digital

Mensajería

- Buzón de entrada común para SMS y MMS
- MMS1.2 para realizar, recibir, editar y transmitir envíos multimedia de hasta 300 kB de tamaño
- Envío de correo electrónico con anexos: soporta el SMTP, los protocolos de POP3 e IMAP4
- Mensajería instantánea
- Envío de mensajes de voz y de clips de sonido vía MMS

Multimedia

- Formatos soportados por el reproductor de música: WMA, MP3, MP4, AAC, AAC+ y eAAC+
- Equipo de Radio FM
- Soporte de sistema de video continuo para el formato de 3GPP
- Hasta 64 tonos midi polifónicos
- Lanzamiento de DRM (la dirección de derechos digital)1.0

Memoria

- 11 MB de memoria libre interna
- Capacidad de memoria expansible con la tarjeta de memoria de MicroSD de hasta 2 GB

Aplicaciones

- Java MIDP 2.0 para juegos y aplicaciones

Conectividad

- Near Field Communication (NFC) con lectura, escritura y capacidad para intercambio de información peer-to-peer
- Chip integral seguro ,la plataforma global2.1.1 almacena la información personal
- Suite de PC de Nokia con USB, Bluetooth, y conectividad de IrDa
- Versión de Bluetooth2.0
- Sincronización de datos de SyncML de transmisión local en exteriores
- Conector de ™ para música
- Puerto para la conectividad de USB

Buscador

- Buscador XHTML integrado
- Dirección de derechos Digital de OMA1.0

Transferencia de datos

- EDGE (EGPRS): clase de multislot 10
- GPRS: clase de multislot 10
- Bluetooth 2.0

Dirección de Información Personal (PIM)

- Información del tiempo real con un calendario mejorado y la posibilidad de ver las notas de usuario en un nuevo modo de actividad “standby”
- Listas de notas, calculadora, reloj despertador, y reloj automático de cuenta regresiva

Características de la comunicación de Campo Cercano (Near Field Communication (NFC)

- Capacidad de pagos y ticketing
- Acceso a servicios movibles e información de forma simple
- Usa la especificación de Java (257 de JSR) para aplicaciones de NFC

Características de Voz

- Diálogos de voz mejorados con la tecnología SIND(Speaker-independent name dialing)
- Manos-libres integrado de alta calidad para una mejor experiencia auditiva
- Mandatos de voz
- Grabación de voz

Servicios Digitales

- Temas con animados wallpapers y protectores de pantalla, además de sonidos incorporados asociados a acciones.
- Tonos: MP3 y MIDI para alertas, videos, y juegos. Soporte para 64 tonos polifónicos
- Posibilidad de descargas OTA para: temas, tonos midi, tonos mp3, protectores de pantalla, wallpapers, transmisión continua de 3GPP, imágenes y videos, además de paquetes de Java para Series 40, aplicaciones de mejora de chip y servicios de NFC (compras, servicios móviles, etc.)

Energía

Batería	**Tiempo de charla**	**Recurso tiempo**	**Capacidad**
BL - 4C	Hacia arriba hacerlo/serlo3.4 horas	Hasta 240 horas	860 MAh

5.1.2 Tecnología NFC (Near Field Communication)

El NFC es una tecnología de conectividad inalámbrica de corto alcance que brinda la comunicación simple, instintiva y segura entre dispositivos electrónicos. Nokia es activo en el área de NFC. El primer teléfono que introdujo la compañía en el mercado con la tecnología NFC fue precisamente el Nokia 6131 NFC, dispositivo usado en este proyecto, lo hizo en 2007.

NFC es una de las más recientes tecnologías de comunicación inalámbricas. Como una tecnología de conectividad inalámbrica de corto alcance, NFC ofrece la comunicación segura entre dispositivos electrónicos. Usuarios de dispositivos permitidos por NFC sólo pueden señalar con el dedo o tocar sus dispositivos si interactúan con otros elementos permitidos por NFC en el ambiente de la comunicación, haciendo uso de la aplicación y los datos fácil y conveniente.

Uno de los objetivos principales de la tecnología de NFC es beneficiarse de las comunicaciones de corto alcance asequible a los consumidores mundialmente. La base de la tecnología de identificación (RFID) de frecuencia de radio existente ha sido impulsada por empresas por ejemplo de tipo logístico o de seguimiento de envíos.

Con sólo pulsar una tecla NFC permite el uso fácil de los dispositivos y los artilugios que usamos diariamente. He aquí algunos ejemplos de qué puede hacerse con un teléfono móvil de NFC en un ambiente permitido por NFC:

- Descargar música o video.
- Cambiar tarjetas comerciales con otro teléfono.
- Pagar tarifas de autobús o tren.
- Imprimir sobre una impresora.
- Usar una unidad terminal de punto de venta para pagar una compra, de la misma manera que una tarjeta de crédito.
- Emparejar dos dispositivos Bluetooth.

Hasta ahora, las tres facetas de tecla y los casos de uso de NFC han sido el pago y ticketing, servicios de iniciación con funcionalidad de lectura/escritura, y compartimiento de archivos peer-to-peer (P2P). NFC ofrece soporte para pagos y servicios de ticketing habilitando el uso de tickets y tarjetas “contactless” que son almacenados en los dispositivos móviles. Mejor que tener que llevar tarjetas fisicas en los bolsillos. Los consumidores pueden almacenar varias tarjetas en teléfonos habilitados con NFC.

Un teléfono habilitado con funciones NFC funciona como tarjetas inteligentes que son usadas en todo el mundo, como auténticas tarjetas de crédito para transacciones públicas entre sistemas. En cuanto a aplicaciones, como una aplicación de tarjeta de crédito, el teléfono habilitado con NFC permite al cliente pagar sólo agitando el teléfono cerca de un lector de punto de venta. El teléfono de NFC también brinda la seguridad mejorada, permitiendo que el usuario proteja las aplicaciones seguras a través de las características de interfaz de usuario del teléfono.

En el futuro, los usuarios podrán recoger la información de nuestro entorno vital utilizando tecnología de NFC. NFC permite que dispositivos movibles lean la

información guardada en etiquetas de NFC localizadas en objetos diarios, como señales de parada de autobús, placas con los nombres de las calles, medicinas, certificados, y embalaje de comida.

Del árbol principal de casos de uso de la tecnología NFC, dos ya están integrados en tecnologías e infraestructuras existentes. Son los casos del pago y ticketing contactless, que pueden ser desarrollados cada vez de forma más amplia y con nuevos objetivos en el mundo de los servicios con tarjetas inteligentes. Estos dos casos suministran un punto de partida para la adopción de la tecnología NFC que minimiza los costes iniciales. El tercer caso de uso, el compartimiento (P2P), todavía está por venir. Está previsto que el P2P combinado con tecnologías de etiquetado dará forma al mercado hacia tecnologías mas uniformes y abiertas.

Existe un foro de debate de NFC. Fue formado para avanzar en el uso de tecnología de NFC. Sobre sus objetivos se encuentran el desarrollo de las especificaciones, asegurar la interoperabilidad entre dispositivos de NFC y servicios, y educar el mercado sobre la tecnología de NFC estar. El foro de debate, constituido en 2004, tiene más de 120 miembros ahora. Fabricantes, desarrolladores de aplicaciones, instituciones del servicio de gestión financiera, y otros han estado cooperando para promover el uso de la tecnología de NFC en electrónica de consumo para dispositivos móviles y PCs.

5.1.3 Plataforma Series 40

La plataforma de Series 40 es la plataforma para dispositivos móviles más usada en el mundo. La serie 40 fue la primera en otorgar a los dispositivos móviles de acceso a Internet y/o a diferentes sectores específicos del mercado como la música o la moda.

Suministra un juego común de tecnologías y APIs para el desarrollo de aplicaciones y la distribución de contenidos digitales. La plataforma incluye un juego de APIs de Java, un ambiente de búsqueda, servicio de intercambio de mensajes de multimedia (MMS), capacidad de temas y wallpapers, y soporte para el audio y video, ambos con posibilidad de descarga y streaming.

5.1.3.1 Tecnologías soportadas

La plataforma S40 trabaja con tecnología Java, Flash Lite de Adobe, tecnología Web, contenidos multimedia,…Además, para los desarrolladores de Java, existe la tecnología MIDP y CLDC, que junto con los paquetes de JSRs oportunos, provee al dispositivos de más herramientas y aplicaciones para las comunicaciones, mensajería, multimedia y capacidad de gráficos. Por ejemplo el interfaz para desarrollo de aplicaciones (API) JSR-82 nos facilita librerias, métodos, funciones y clases para desarrollar aplicaciones Java que hagan uso de Bluetooth.

Una amplia gama de dispositivos que incluyen la plataforma de Series 40 están disponibles para la mayoría de los estándares inalámbricos, incluyendo GSM, CDMA, EDGE, y sistema de telecomunicaciones móviles universal de banda ancha CDMA (WCDMA) y UMTS. Además, los dispositivos de Series 40 ofrecen opciones de

conectividad de red local, incluyendo tecnología inalámbrica Bluetooth, USB, IrDA y WLAN usando la tecnología de acceso para móviles (UMA).

5.1.3.2 Tecnologías de desarrollo

La plataforma S40 ofrece dos opciones de desarrollo para crear aplicaciones que puedan ser instaladas en un dispositivo: la plataforma Java, tecnología micro edition (Java ME), con configuración CLDC 1.1 y el perfil de información de dispositivo móvil (MIDP) 2.0 o 2.1.
Además, aplicaciones basadas en tecnología web puede ser creadas usando: HTML 4.01, CSS2, JavaScript 1.5, y WML 2.0 con soporte para WMLScript.

Programar código directamente sobre el sistema operativo no es posible.

5.1.3.3 Descarga e instalación de aplicaciones

Otras aplicaciones pueden ser descargadas Over-The-Air (OTA) e instaladas directamente en el dispositivo S40. Aunque también pueden ser descargadas desde Internet a un PC y desde allí instalarse en el dispositivo S40 usando el "Nokia PC Suite" vía USB, Bluetooth o con una conexión IrDA.

5.1.3.4 Mecanismos de configuración del dispositivo

La plataforma de Series 40 soporta tres tecnologías que permiten la configuración del dispositivo. Éstas son:

-Open Mobile Alliance (OMA) Client Provisioning (CP) de Mobile (CP).
Mecanismo que permite que ajustes sean enviados a un dispositivo como un XML (formato de mensaje SMS). Estos mensajes pueden ser enviados al dispositivo o alojados en una tarjeta SIM.

-OMA Device Management (DM). Mecanismo que permite que los ajustes se mantengan en el dispositivo sobre una conexión HTTP. Las actualizaciones de configuración pueden ser iniciadas tanto por un servidor definido como por el usuario del dispositivo. OMA DM también proporciona un mecanismo para iniciar actualizaciones del dispositivo S40 over-the-air del firmware de OMA (FOTA).

-Servicios móviles Plug and Play (PnP-MS). Este mecanismo abre el navegador del dispositivo en una página desde la que el usuario puede pedir los detalles de configuración de CP de OMA.

5.1.3.5 Estándares de Audio y Video de la plataforma Series 40

Además de archivos de sonido en MIDI y tonos True, desde la 3ª edición de la Series 40 en adelante, los dispositivos soportan audio en streaming (AMR-WB) y codificación de audio avanzada (AAC).
La plataforma de Series 40 soporta video en formatos H.263 y MPEG-4. Y desde la 3ª edición en adelante también soporta video en streaming en formato H.263. Además desde la 6ª edición e adelante también soporta el formato de Windows Media Video (WMV)

5.1.3.6 Perfiles MIDP y versiones de JSRs de Java soportados por S40

1ª Edición

La 1ª edición de Series 40 soporta la plataforma Java™, Java Micro Edition (Java™ ME), con configuración de dispositivos conectados limitada (CLDC) 1.0, y tecnología de información de perfil de dispositivo móvil (MIDP) 1.0. Además, la API de Nokia UI, proporciona a los programadores más capacidad en áreas de gráficos, trazados en pantalla completa, acceso a softkeys y configuraciones de sonidos y vibraciones. Algunas versiones de la plataforma S40 también soportan la API de mensajería inalámbrica (JSR-120).

2ª Edición

Series 40 2ª Edición obedece a la tecnología Java™ para la industria inalámbrica (JSR-185) y soporta CLDC 1.1 y MIDP 2.0. La 2ª edición también añade soporte para tres APIs más:

-APIs java para Bluetooth (JSR-82). El protocolo estándar Bluetooth RFCOMM es utilizado para descubrir el servicio.

-Wireless Messaging API (JSR-120), que proporciona acceso independiente a SMS

-Mobile Media API (JSR-135), que proporciona a los desarrolladores acceso y configuración hacia los tonos MIDI que son reproducidos en dispositivos S40.

-Además, en algunos dispositivos de la series 40 pueden incluirse los paquetes opcionales de PDA (JSR-75) para la plataforma J2ME y la API de Mobile 3D Graphics (JSR-184) como es el caso del nokia 6230i imaging phone.

3ª Edición

La 3ª edición de la plataforma Series 40 es la que contiene el teléfono de nuestro proyecto 6131 NFC, esta edición conserva JSR-185, construida en CLDC 1.1 y tiene soporte para MIDP 2.0.

Está basada en características de la 2ª edición incluyendo una mejora en la API Mobile Media (JSR-135), con soporte tanto como para reproducción de muestras de sonido como para imágenes y videos.

Se incluyen también paquetes opcionales para PDAs, incluyendo PIM (Personal Information Management) y FC (FileConnection), que permiten a las aplicaciones acceder al calendario, lista de contactos, notas de recordatorio, además de llevar a cabo tareas como la de salvar e-mails.

Una API llamada Mobile 3D Graphics (JSR-184), facilita características para crear gráficos mejorados en 3D para juegos, mensajes animados, interfaces de usuario y visualizaciones interactivas de productos.

Existe un paquete llamado “Feature Pack 1” que añade a la 3ª Edición de Series 40 soporte para las siguientes JSRs de Java:

-Wireless Messaging API (JSR-205), que soporta el envío y recibo de SMS y MMS vía GSM.

-Scalable 2D Vector Graphics API para J2ME™ (JSR-226), que permite representar imágenes de vectores en 2D escalables, incluyendo imágenes externas en formato SVG. Los usos principales para esta API se encuentran en la visualización de mapas, iconos escalables y aplicaciones que requieren escalado y gráficos animados.

El paquete “Feature Pack 2” añade soporte para el protocolo: “Application Protocol Data Unit” (APDU), API de seguridad y servicios de confianza (JSR-177) para J2ME.

5ª Edición

La siguiente edición que lanzaron los desarrolladores de Nokia fue directamente la 5ª, sin pasar por la 4ª (para evitar la redundancia de cuatros). Esta 5ª edición de la plataforma Series 40 añade soporte para el MIDP 2.1 (JSR-118) y el subconjunto de “Mobile Service Architecture” (JSR-248). JSR-248 está diseñado para reducir la fragmentación Java definiendo un grupo consistente de tecnologías Java para un gran volumen de dispositivos móviles. El subconjunto abarca JSR-75, JSR-82, JSR-118, JSR-135, JSR-184, JSR-205, y JSR-226.

Además, la 5ª edición de Series 40 provee de las siguientes nuevas APIs o actualizaciones de Java™ :

-APIs para Bluetooth v1.1 (JSR-82), con la adición de soporte para protocolo OBEX.

-Especificaciones de servicios Web J2ME™ (JSR-172), que ahora implementa la API de Java para llamada basada en procedimiento remoto XML (JAX-RPC) , que permite a las aplicaciones usar SOAP para acceder de forma pública o privada a los servicios Web.

-API de Seguridad y Servicios de Confianza (JSR-177), que ahora incluyen el paquete opcional SATSA-CRYPTO, que permite a las aplicaciones ofrecer características de criptografía.

-Suplementos de multimedia avanzados (JSR-234), que proveen de audio 3D y música y soporte para otorgar a las aplicaciones de una mejor experiencia auditiva para juegos y aplicaciones multimedia.

Con el paquete “Feature Pack 1” para esta 5ª edición de Series 40 se añadieron las siguientes nuevas características y mejoras para los APIs de Java:

-Paquete opcional de PDA para J2ME (JSR-75), que ahora incluye soporte para campo de contacto de video por URL.

-Mobile Media API (JSR-135), que ahora puede manejar reproducción progresiva de audio y video y producir mezclas de audio.

-Content Handler API (JSR-211), que permite a los MIDlets ser especificados por los manejadores por uno o más tipos de archivo, así se permite que las aplicaciones manejen contenidos multimedia y web de manera uniforme.

-Suplementos de multimedia avanzados (JSR-234), que ahora podrán realizar mezclas de audio, incluyendo mezclas de audio en 3D.

6ª Edición

Con la Series 40 6ª edición los desarrolladores de Nokia añaden, entre otras JSRs ,el soporte de arquitectura para servicios móviles JSR-248, que no sólo ofrece APIs para plataformas Nokia sino también un grupo de APIs comunes para un amplio rango de teléfonos móviles de otros fabricantes que usen Java. También añade nuevas API de Java mejoradas:

-API de ubicación para J2ME ™ (JSR-179), que permite que aplicaciones adquieran información de ubicación, escuchen cambios en la ubicación, e identifiquen proximidades a una ubicación usando un GPS incorporado en un dispositivo o conectado por Bluetooth.

-API de mensajería inalámbrica (JSR-205) con la inclusión del soporte para el servicio de transmisión de celda (CBS).

5.2 Plataforma Java Micro Edition (J2ME)

J2ME es el acrónimo de Java 2 Micro Edition. J2ME es la versión de Java orientada a los dispositivos móviles. Debido a que los dispositivos móviles tienen una potencia de cálculo baja e interfaces de usuario pobres, es necesaria una versión específica de Java destinada a estos dispositivos, ya que el resto de versiones de Java, J2SE o J2EE, no encajan dentro de este esquema. J2ME es por tanto, una versión "reducida" de J2SE.

5.2.1 Configuración

La configuración es un mínimo grupo de APIs (Application Program Interface), útiles para desarrollar las aplicaciones destinadas a un amplio rango de dispositivos. La configuración estándar para los dispositivos inalámbricos es conocida como **CLDC** (Connected Limited Device Configuration). El CLDC proporciona un nivel mínimo de funcionalidades para desarrollar aplicaciones para un determinado conjunto de dispositivos inalámbricos. Se puede decir que CLDC es el conjunto de clases esenciales para construir aplicaciones. Hoy por hoy, sólo tenemos una configuración, pero es de esperar que en el futuro aparezcan distintas configuraciones orientadas a determinados grupos de dispositivos. Los requisitos mínimos de hardware que contempla CLDC son:

- 160KB de memoria disponible para Java
- Procesador de 16 bits
- Consumo bajo de batería
- Conexión a red

Los dispositivos que claramente encajan dentro de este grupo son, los teléfonos móviles, los PDA (Personal Digital Assintant), los "Pocket PC"...

En cuanto a los requisitos de memoria, según CLDC, los 160KB se utilizan de la siguiente forma:

- 128KB de memoria no volátil para la máquina virtual Java y para las librerías del API de CLDC
- 32KB de memoria volátil, para sistema de ejecución (Java Runtime System).

En cuanto a las limitaciones impuestas por CLDC, tenemos por ejemplo las operaciones en coma flotante. CLDC no proporciona soporte para matemática en coma flotante. Otra limitación es la eliminación del método *Object.finalize*. Este método es invocado cuando un objeto es eliminado de la memoria, para optimizar los recursos. También se limita el manejo de las excepciones. Es complicado definir una serie de clases de error estándar, que se ajuste a todos los dispositivos contemplados dentro de CLDC. La solución es soportar un grupo limitado de clases de error y permitir que el API específico de cada dispositivo defina su propio conjunto de errores y excepciones.

La seguridad dentro de CLDC es sencilla, sigue el famoso “modelo sandbox”. Las líneas básicas del modelo de seguridad sandbox en CLDC son:

- Los ficheros de clases, deben ser verificados como aplicaciones válidas.
- Sólo las APIs predefinidas dentro de CLDC están disponibles.
- No se permite cargadores de clases definidos por el usuario.
- Sólo las capacidades nativas proporcionadas por CLDC son accesibles.

5.2.2 Perfiles

En la arquitectura de J2ME, por encima de la configuración, tenemos el perfil (profile). El perfil es un grupo más específico de APIs, desde el punto de vista del dispositivo. Es decir, la configuración se ajusta a una familia de dispositivos, y el perfil se orienta hacia un grupo determinado de dispositivos dentro de dicha familia. El perfil, añade funcionalidades adicionales a las proporcionadas por la configuración. La especificación **MIDP** (Mobile Information Device Profile), describe un dispositivo MIDP como un dispositivo, pequeño, de recursos limitados, móvil y con una conexión “inalámbrica”.

5.2.3 MIDLet

Las aplicaciones J2ME desarrolladas bajo la especificación MIDP, se denominan MIDLets. Las clases de un MIDLet, son almacenadas en códigos .java, dentro de un fichero .class. Estas clases, deben ser verificadas antes de su “puesta en marcha”, para garantizar que no realizan ninguna operación no permitida. Este preverificación, se debe hacer debido a las limitaciones de la máquina virtual usada en estos dispositivos. Esta máquina virtual se denomina KVM.

Los MIDLets, son empaquetados en ficheros “.jar”. Se requiere alguna información extra, para la puesta en marcha de las aplicaciones. Esta información se almacena en el fichero de “manifiesto”, que va incluido en el fichero “.jar” y en un fichero descriptor, con extensión “.jad”. Un fichero “.jar” típico, por tanto, se compondrá de:

- Clases del MIDLet
- Clases de soporte
- Recursos (imágenes, sonidos...)
- Manifiesto (fichero “.mf”)
- Descriptor (fichero “.jad”)

Todos los MIDLets, deben heredar de la clase javax.microedition.midlet.MIDlet, contenida en el API MIDP estándar. Esta clase define varios métodos, de los cuales destacan los siguientes:

o startApp() – Lanza el MIDLet
o pauseApp() – Para el MIDLet
o destroyApp() – Detruye el MIDLet

Ciclo de Vida

Los MIDLets tienen tres posibles estados que determinan su comportamiento: activo, parado y destruido. Estos estados se relacionan directamente con los métodos antes enumerados. Estos métodos son llamados directamente por el entorno de ejecución de aplicaciones, pero también pueden ser invocados desde código.
Los métodos indicados anteriormente, serán normalmente utilizados para gestionar los recursos (solicitar y liberar).
Un MIDLet puede ser lanzado y parado varias veces, pero sólo será destruido una vez.

Comandos Midlet

La mayoría de los MIDLets, implementan el método *commandAction()*, un método de respuesta a eventos, definido en la interfaz *javax.microedition.lcdui.CommandListener*. La forma de funcionar de este método, es similar al control de eventos típico de java.

Display, Screen y Canvas

La clase javax.microedition.lcdui.Display, representa el controlador de pantalla del dispositivo. Esta clase es responsable de controlar la pantalla y la interacción con el usuario. Nunca se debe crear un objeto Display, normalmente se obtiene una referencia al objeto Display en el constructor del MIDLet, y se usa para crear la interfaz de usuario en el método startApp(). Hay una instancia de Display por cada MIDLet que se está ejecutando en el dispositivo. Mientras que del objeto Display sólo hay una instancia, del objeto javax.microedition.lcdui.Screen, pueden existir varias. El objeto Screen, es un componente GUI genérico, que sirve como base para otros componentes. Estos objetos representan una pantalla entera de información. Sólo se puede mostrar una pantalla cada vez. La mayoría de los MIDLets, usan subclases de la clase Screen, como Form, TextBox o List, ya que proporcionan una funcionalidad mucho más específica.

Una clase similar en cierta medida a Screen, es la clase Canvas, perteneciente al mismo paquete. Los objetos Canvas son usados para realizar operaciones gráficas directamente, como puede ser pintar líneas o imágenes. No se puede mostrar un objeto Canvas y un objeto Screen a la vez, pero se pueden alternar en la misma aplicación.

5.2.4 Las APIs de CLDC y de MIDP

Los elementos principales involucrados en el proceso de desarrollo con Java ME, son el lenguaje Java ME propiamente dicho y el grupo de APIs (Application Programming Interface) que proporcionan el soporte para el software desarrollado.

Los APIs específicos para el desarrollo de MIDlets, están descritos en las especificaciones de CLDC y MIDP, que definen la configuración y el perfil para los dispositivos inalámbricos móviles. Estas APIs constan de clases e interfaces ya presentes en el estándar J2SE así como con clases e interfaces únicos del desarrollo de MIDlets. Como ya se ha mencionado, los MIDlets son aplicaciones especiales, diseñadas bajo los requerimientos de la especificación MIDP. Esta especificación son una serie de normas que indican las capacidades y restricciones de Java respecto a los dispositivos móviles. Un aspecto importante de estas capacidades y limitaciones,

es el conjunto de clases e interfaces disponibles para afrontar el desarrollo de aplicaciones.

La especificación MIDP provee una descripción detallada del API disponible para el desarrollo de MIDlets. El CLDC proporciona un API adicional. De hecho, el API de MIDP, se basa en el API de CLDC, para construir clases e interfaces más específicos.

El API de CLDC

El API de CLDC es un pequeño subgrupo del API de J2SE. A parte de estas clases e interfaces, el API de CLDC contiene una serie de interfaces propias, dedicadas a los servicios de red.

-El paquete java.lang

Las clases e interfaces del paquete java.lang, están relacionadas con el núcleo del lenguaje Java. Es decir, estás clases incluyen soporte para las capacidades del lenguaje como los recubrimientos de los tipos primitivos de variables, las cadenas, la excepciones y los threads (hilos), entre otras.
Las clases e interfaces del lenguaje java.lang son:

- Boolean – Encapsula el tipo primitivo boolean
- Byte – Encapsula el tipo primitivo byte
- Character – Encapsula el tipo primitivo char
- Class – Proporciona información sobre la ejecución de una clase
- Integer – Encapsula el tipo primitivo int
- Long – Encapsula el tipo primitivo long
- Math – Proporciona acceso a varias operaciones y constantes matemáticas
- Object – La superclase del resto de clases en Java
- Runnable – Interfaz que proporciona un significado a la creación de threads, sin heredar la clase Thread
- Runtime – Proporciona acceso al entorno de ejecución
- Short – Encapsula el tipo primitivo short
- String – Representa una cadena de texto constante
- StringBuffer – Representa una cadena de texto, de longitud y valor variable
- System – Proporciona acceso a los recursos del sistema
- Thread – Se usa para crear un “thread” (hilo) de ejecución dentro de un programa
- Throwable – Proporciona soporte para el control de excepciones

- El paquete java.util

El paquete java.util, como en J2SE, incluye clases e interfaces con utilidades variadas, como puede ser el manejo de fechas, estructuras de datos...
Las clases e interfaces de java.util son:

- Calendar – Proporciona funciones para manejar fechas y convertir valores numéricos en fechas
- Date – Representa un instante de tiempo

- Enumeration – Es una interfaz que describe como manejar la iteración entre un grupo de valores
- Hashtable – Una colección que asocia valores y claves
- Random – Generador de números pseudoaleatorios
- Stack – Una colección con gestión LIFO
- TimeZone – Representa la zona horaria
- Vector – Una colección en forma de matriz dinámica

-El paquete java.io

Este paquete proporciona clases e interfaces de apoyo para leer y escribir datos. Aunque la funciones de persistencia, recaen sobre los perfiles.
Las clases e interfaces de java.io son:

- ByteArrayInputStream – Un flujo (stream) de entrada que se gestiona internamente como una matriz de bytes
- ByteArrayOutputStream – Un flujo (stream) de salida que se gestiona internamente como una matriz de bytes
- DataInput – Una interfaz que define los métodos para leer datos desde un flujo (stream) binario a tipos primitivos
- DataInputStream -Un flujo (stream) desde el cual se leen datos como tipos primitivos
- DataOutput – Una interfaz que define métodos para escribir datos en forma de tipos primitivos en un flujo (stream) binario
- DataOutputStream – Escribe datos en tipos primitivos en un flujo (stream) en formato binario
- InputStream – La clase base para todos los flujos (streams) de entrada
- InputStreamReader – Un flujo (stream) desde el que se pueden leer caracteres de texto
- OutputStream – La clase base para todos los flujos (streams) de salida
- OutputStreamWriter – Un flujo (stream) en el que se pueden escribir caracteres de texto
- PrintStream – Un flujo (stream) de escritura que facilita el “envío” de datos en forma de tipos primitivos
- Reader – Una clase abstracta para leer flujos (streams) de lectura
- Writer – Una clase abstracta para leer flujos (streams) de escritura

El API de MIDP

El perfil de dispositivo comienza donde la configuración para, en lo que se refiere a proveer funciones, llevar a cabo importantes tareas en un determinado tipo de dispositivo. De forma similar a lo que hicimos con el API de CLDC, el API de MIDP se puede dividir en dos partes. Dos clases heredadas directamente del API de J2SE y una serie de paquetes que incluyen clases e interfaces únicas para el desarrollo de MIDP.

- Las clases heredadas de J2SE

Sólo dos clases del API de MIDP, provienen directamente del API de J2SE. Estas clases están dentro del paquete java.util, dentro del API de MIDP.

-Timer – Proporciona funcionalidad para crear tareas programadas temporalmente
-TimerTask – Representa una tarea que es temporizada a través de la clase Timer

- Clases e interfaces propios de MIDP

La gran parte del API de MIDP, son una serie de clases e interfaces diseñadas explícitamente para la programación de MIDLets. Aunque estas clases e interfaces son similares a algunas clases del API de J2SE, son totalmente exclusivas del API de MIDP. Esta parte del API se divide en varios paquetes:

- javax.microedition.midlet
- javax.microedition.lcdui
- javax.microedition.io
- javax.microedition.rms

- El paquete javax.microedition.midlet

Este es el paquete central del API de MIDP y contiene una sola clase:
MIDlet. Esta clase provee la funcionalidad básica para que una aplicación se pueda ejecutar dentro de un dispositivo con soporte para MIDLets.

- El paquete javax.microedition.lcdui

Este paquete contiene clases e interfaces que soportan componentes de interfaz de usuario (GUI), específicos para las pantallas de los dispositivos móviles.

Lcdui, corresponde al texto Liquid Crystal Displays User Interfaces.
Las pantallas de cristal líquido son comunes en los dispositivos móviles.

Un concepto básico dentro de la programación de MIDLets, es la pantalla (screen), que es un componente GUI genérico, que sirve de clase base para otros componentes. Las interfaces de usuario, se compondrán añadiendo componentes a la clase base (screen). A parte de la creación de interfaces de usuario mediante esta aproximación de alto nivel, también se puede acceder directamente a primitivas de dibujo, sobre la pantalla. En este caso, la superficie de la pantalla del dispositivo es como un lienzo (canvas).

Las interfaces del paquete javax.microedition.lcdui son:

- Choice – Una interfaz que describe una serie de elementos sobre los que el usuario puede escoger
- CommandListener – Una interfaz de monitorización de eventos (listener), para gestionar comandos a alto nivel
- ItemStateListener – Una interfaz de monitorización de eventos (listener) para gestionar los eventos sobre el estado de los elementos

Además de las interfaces antes enumeradas, el paquete lcdui, contiene también las siguientes clases:

- Alert – Una pantalla que muestra información al usuario y después desaparece.
- AlertType – Representa diferentes tipos de alertas, usadas junto con la clase Alert
- Canvas – Una superficie (lienzo) para dibujar a bajo nivel. Permite dibujar las pantallas que mostrará el dispositivo, a bajo nivel
- ChoiceGroup – Presenta un grupo de elementos seleccionables. Se usa junto con el interfaz Choice
- Command – Representa un comando a alto nivel, que puede ser generado desde el MIDLet
- DateField – Representa una fecha y una hora que pueden ser editadas
- Display – Representa la pantalla del dispositivo y acoge la recuperación de las acciones del usuario
- Displayable – Es un componente abstracto que se puede mostrar por pantalla. Es una superclase para otros componentes.
- Font – Representa un tipo de letra y las métricas asociadas al mismo
- Form – Es una pantalla que sirve como contenedor para otros componentes gráficos de usuario
- Gauge – Muestra un valor, como un porcentaje dentro de una barra
- Graphics – Encapsula operaciones gráficas bidimensionales, como son el dibujo de líneas, elipses, texto e imágenes
- Image – Representa una imagen
- ImageItem – Es un componente que soporta la presentación (layout) de una imagen
- Item – Es un componente que representa un elemento con una etiqueta List – Es un componente que consiste en una lista de opciones para seleccionar
- Screen – Representa una pantalla completa a alto nivel, y sirve como clase base para todos los componentes del interfaz de usuario de MIDP
- StringItem – Un componente que representa un elemento consistente en una etiqueta y una cadena de texto asociada
- TextBox – Un tipo de pantalla que soporta la visualización y edición de texto
- TextField – Un componente que soporta la visualización y edición de texto. A diferencia de un TextBox, este componente puede ser añadido a un Form, junto con otros componentes
- Ticker – Un componente que muestra texto moviéndose horizontalmente, como una marquesina

- El paquete javax.microedition.io

El CLDC, descarga el trabajo con la red y la entrada/salida en al paquete java.io y en el Generic Connection Framework (GCF). El API de MIDP parte de esta base, añadiendo la interfaz *HttpConnection*, que pertenece al paquete javax.microedition.io.

- El paquete javax.microedition.rms

El API de MIDP, presenta un sistema de persistencia basado en registros para almacenar información. Este sistema, conocido como Record Management System (RMS).
Las interfaces del paquete javax.microedition.rms son:

- RecordComparator – Para comparar dos registros
- RecordEnumeration – Para iterar sobre los registros
- RecordFilter – Para filtrar registros de acuerdo a un registro
- RecordListener – Un monitorizador de eventos usado para controlar los cambios en los registros

A parte de estas interfaces, tenemos una serie de clases, de las que debemos destacar la clase RecordStore, que representa un record store (almacén de registros).
Las clases del paquete javax.microedition.rms son:

- InvalidRecordException – Se lanza cuando una operación falla porque el identificador del registro es invalido
- RecordStore – Representa un "almacén de registros"
- RecordStoreException – Se lanza cuando una operación falla por un error general
- RecordStoreFullException - Se lanza cuando una operación falla porque el record store está completo
- RecordStoreNotFoundException – Se lanza cuando una operación falla porque el record store no se ha podido localizar
- RecordStoreNotOpenException – Se lanza cuando se realiza una operación sobre un record store cerrado

5.2.5 Persistencia de Datos (RMS)

Es de suponer que para la gestión de la información del grupo de contadores es necesario el uso de cierto tipo de memoria para almacenar y después recuperar los datos oportunos y según vayan pidiéndolos las pantallas que hemos creado para nuestro software de monitorización.

El sistema de gestión de registros (Record Management System, RMS) se compone de una serie de clases e interfaces que proporcionan soporte a un sistema simple de base de datos que es usado para almacenar esta información que nos llega de la estación base Bluetooth. Es de esperar que la información que maneje el MIDlet no sea excesivamente complicada.

El objetivo del RMS es almacenar datos de tal forma que estén disponibles una vez que el MIDlet pare su ejecución. La unidad básica de almacenamiento es el registro (record) que será almacenado en un base de datos especial, denominada almacén de registros (record store). Cuando un MIDlet usa un almacén de registros, primero debe crearlo y luego añadir los registros. Cuando un registro es añadido a un almacén de registros, se le asigna un identificador único (record id).

El paquete RMS

El API de MIDP incluye un paquete para el RMS, javax.microedition.rms. Este paquete incluye clases e interfaces que proporcionan un marco de trabajo para los registros, los almacenes y otras características. Básicamente tenemos:

- Capacidad para añadir y borrar registros de un almacén
- Capacidad para compartir almacenes por parte de todos los MIDlets de un MIDlet suite

Para representar el record store tenemos la clase *RecordStore*. A parte de esta clase, tenemos varias interfaces que presentan la funcionalidad para enumerar registros y filtrarlos.

La clase RecordStore

Esta clase nos permite abrir, cerrar y borrar almacenes de registros. También podemos añadir, recuperar y borrar registros, así como a enumerar los registros de un almacén. Los métodos son:

- openRecordStore – Abre el almacén de registros
- closeRecordStore – Cierra el almacén de registros
- deleteRecordStore – Borra el almacén de registros
- getName – Recupera el nombre del almacén de registros
- getNumRecords – Recuperar el número de registros del almacén
- addRecord – Añade un registro al almacén de registros
- getRecord – Recupera un registro del almacén de registros
- deleteRecord – Borra un registro del almacén de registros
- enumerateRecord – Obtiene una enumeración del almacén de registros

Las interfaces de RMS

A parte de la clase RecordStore, tenemos las siguientes interfaces:

- RecordEnumeration – Describe una enumeración del almacén de registros
- RecordComparator – Describe la comparación de registros
- RecordFilters – Describe como usar filtros sobre los registros
- RecordListener – Describe un listener que recibe notificaciones cuando un registro es añadido, modificado o borrado del almacén de registros

Abrir un almacén de registros

El primer paso para trabajar con RMS es abrir el almacén de registros usando el método estático openRecordStore de la clase RecordStore. Con este método también podemos crear un almacén nuevo y abrirlo. El siguiente código muestra como crear un almacén y abrirlo:

```
RecordStore rs = RecordStore.openRecordStore("data", true);
```

El primer parámetro indica el nombre del almacén y el segundo indicará si el almacén debe ser creado o si ya existe. El nombre del almacén debe ser menor de 32 caracteres.

Añadir nuevos registros

Para añadir un nuevo registro a un almacén, debemos tener antes la información en el formato correcto, es decir, como una matriz de bytes. El siguiente código muestra como añadir un registro:

```
int id = 0;
try{
id = recordStore.addRecord(bytes, 0, bytes.length);
}catch (RecordStoreException e){
e.printStackTrace();
}
```

El primer parámetro es la matriz de bytes, el segundo es el desplazamiento dentro de la matriz y el tercero es el número de bytes a añadir. En el ejemplo anterior, añadimos toda la matriz de datos al almacén de registros. El método addRecord devuelve el identificador del registro añadido, que lo identifica unívocamente en el almacén.

Recuperar registros

Para recuperar un registro de un almacén, debemos utilizar el método getRecord, que recupera el registro del almacén a través de su identificador. La información devuelta por este método es una matriz de bytes. Un ejemplo de uso sería:

```
byte[] recordData = null;
try{
recordData = recordStore.getRecord(id);
}catch (RecordStoreException ex){
ex.printStackTrace();
}
```

El código anterior asume que conocemos el identificador del registro.

Borrando registros

De forma similar a cómo se recuperan registros, se pueden borrar. Para borrar un registro, debemos conocer su identificador. El método para borrar un registro es deleteRecord.

Un ejemplo de uso sería:

```
try{
recordStore.deleteRecord(id);
}catch (RecordStoreException ex){
ex.printStackTrace();
}
```

Cuando se borra un registro, se elimina definitivamente del almacén.

Enumeración de registros

Para enumerar los registros que hay en un almacén, disponemos del método enumerateRecords. Este método nos devuelve un objeto que implementa la interfaz RecordEnumeration, que es lo que usaremos para enumerar los registros. El siguiente ejemplo nos muestra cómo podemos obtener una enumeración de registros:

RecordEnumerationo records =
recordStore.enumerateRecords(null, null, false);

Los parámetros del método son usados para personalizar el proceso. El primer parámetro es un objeto de tipo RecordFilter que es usado para filtrar los registros de la enumeración. El segundo parámetro es un objeto RecordComparator que determina el orden en que los registros son enumerados. El último parámetro es de tipo boolean y determina si la enumeración debe ser actualizada con el almacén. Si este último valor es true, la enumeración será automáticamente actualizada para reflejar los cambios en el almacén, así como las inserciones y eliminaciones en el mismo.

La interfaz RecordEnumeration define algunos métodos necesarios para el manejo de la enumeración. Por ejemplo, el método hasNextElement comprueba si hay más registros disponibles. Para movernos por los registros, podemos usar los métodos nextRecord y nextRecordId. El primero de ellos retorna una matriz de bytes y el segundo retorna el identificador del registro. El siguiente ejemplo recorre el almacén, mostrando los identificadores de los registros:

```
RecordEnumeration records = null;
try{
records = recordStore.enumerateRecords(null, null, false);
while(records.hasNextElement()){
int id = records.getRecordId();
System.out.println(id);
}
}catch (Exception ex){
ex.printStackTrace();
}
```

Cierre de un almacén de datos

Es importante cerrar el almacén una vez que hemos acabado de trabajar con él. La clase RecordStore proporciona el método closeRecordStore con este fin. Este método tiene la siguiente forma:

recordStore.closeRecordStore();

5.3 API JSR-82 para la comunicación Bluetooth

Este API está dividida en dos partes: el paquete javax.bluetooth y el paquete javax.obex. Los dos paquetes son totalmente independientes.

El primero de ellos define clases e interfaces básicas para el descubrimiento de dispositivos, descubrimiento de servicios, conexión y comunicación. La comunicación a través de javax.bluetooth es a bajo nivel: mediante flujos de datos o mediante la transmisión de arrays de bytes. Es la comunicación que necesitamos que realicen los dispositivos de nuestro proyecto, por lo tanto, el paquete javax.bluetooth es el que se ha utilizado para la realización del MIDlet, concretamente en la clase principal llamada "BluetoothWSNMidlet" y que se encarga principalmente del descubrimiento de servicios Bluetooth, de la conexión entre la estación base y el teléfono y del recibimiento de datos

Por el contrario el paquete javax.obex permite manejar el protocolo de alto nivel OBEX (OBject EXchange). Este protocolo es muy similar a HTTP y es utilizado sobre todo para el intercambio de archivos. El protocolo OBEX es un estándar desarrollado por IrDA y es utilizado también sobre otras tecnologías inalámbricas distintas de Bluetooth.

El paquete javax.bluetooth

En una comunicación Bluetooth existe un dispositivo que ofrece un servicio (servidor), en nuestro proyecto la estación base y otros dispositivos acceden a él (clientes), como el teléfono móvil. Dependiendo de qué parte de la comunicación debamos programar deberemos realizar una serie de acciones diferentes.
Un cliente Bluetooth deberá realizar las siguientes:

- Búsqueda de dispositivos. La aplicación realizará una búsqueda de los dispositivos Bluetooth a su alcance que estén en modo "conectable".
- Búsqueda de servicios. La aplicación realizará una búsqueda de servicios por cada dispositivo.
- Establecimiento de la conexión. Una vez encontrado un dispositivo que ofrece el servicio deseado nos conectaremos a él.
- Comunicación. Ya establecida la conexión podremos leer y escribir en ella.

Por otro lado, un servidor Bluetooth deberá hacer las siguientes operaciones:

- Crear una conexión servidora
- Especificar los atributos de servicio
- Abrir las conexiones clientes

Clases básicas

-Clase LocalDevice

Un objeto LocalDevice representa al dispositivo local. Este objeto será el punto de partida de prácticamente cualquier operación.
Alguna información de interés que podemos obtener a través de este objeto es, por ejemplo, la dirección Bluetooth de nuestro dispositivo, el apodo o "friendly-name" (también llamado "Bluetooth device name" o "user-friendly name"). A través de este objeto también podemos obtener y establecer el modo de conectividad: la forma en que nuestro dispositivo está o no visible para otros dispositivos. Esta clase es un "singleton"; para obtener la única instancia existente de esta clase llamaremos al método getLocalDevice() de la clase LocalDevice.

-Excepción BluetoothStateException

Al llamar al método getLocalDevice() se puede producir una excepción del tipo BluetoothStateException. Esto significa que no se pudo inicializar el sistema Bluetooth.

-Clase DeviceClass

El método getDeviceClass() devuelve un objeto de tipo DeviceClass. Este tipo de objeto describe el tipo de dispositivo. A través de sus métodos podremos saber, por ejemplo, si se trata de un teléfono, de un ordenador,...

-Clase UUID

La clase UUID (*universally unique identifier*) representa indentificadores únicos universales. Se trata de enteros de 128 bits que identifican protocolos y servicios. Como veremos más adelante un dispositivo puede ofrecer varios servicios. Los UUID servirán para identificarlos. En la documentación de la clase UUID se puede encontrar una tabla con los protocolos y servicios más comunes y sus UUIDs asignadas.

Búsqueda de dispositivos y servicios

Las búsquedas de dispositivos y servicios son tareas que solamente realizarán los dispositivos clientes.

-Clase DiscoveryAgent

Las búsquedas de dispositivos y servicios Bluetooth las realizaremos a través del objeto DiscoveryAgent. Este objeto es único y lo obtendremos a través del método getDiscoveryAgent() del objeto LocalDevice:

DiscoveryAgent discoveryAgent =
LocalDevice.getLocalDevice().getDiscoveryAgent();

Para comenzar una nueva búsqueda de dispositivos llamaremos al método startInquiry(). Este método requiere dos argumentos. El primer argumento es un entero que especifica el modo de conectividad que deben tener los dispositivos a buscar. Este valor deberá ser DiscoveryAgent.GIAC o bien DiscoveryAgent.LIAC. El segundo argumento es un objeto que implemente DiscoveryListener. A través de este último objeto serán notificados los dispositivos que se vayan descubriendo. Para cancelar la búsqueda usaremos el método cancelInquiry().

-La interfaz DiscoveryListener

La interfaz DiscoveryListener tiene los siguientes métodos:

- *public void deviceDiscovered(RemoteDevice rd, DeviceClass c)*
- *public void inquiryCompleted(int c)*
- *public void servicesDiscovered(int transID, ServiceRecord[] sr)*
- *public void serviceSearchCompleted(int transID, int respCode)*

Los dos primeros métodos serán llamados durante el proceso de búsqueda de dispositivos. Los otros dos métodos serán llamados durante un proceso de búsqueda de servicios. Pasemos a investigar más profundamente los dos primeros métodos que son los usados durante una búsqueda de dispositivos.

public void deviceDiscovered(RemoteDevice rd, DeviceClass c)

Cada vez que se descubre un dispositivo se llama a este método. Nos pasa dos argumentos. El primero es un objeto de la clase RemoteDevice que representa el dispositivo encontrado. El segundo argumento nos permitirá determinar el tipo de dispositivo encontrado.

public void inquiryCompleted(int c)

Este método es llamado cuando la búsqueda de dispositivos ha finalizado. Nos pasa un argumento entero indicando el motivo de la finalización. Este argumento podrá tomar los valores:

-DiscoveryListener.INQUIRY_COMPLETED si la búsqueda ha concluido con normalidad.
-DiscoveryListener.INQUIRY_ERROR si se ha producido un error en el proceso de búsqueda.
-DiscoveryListener.INQUIRY_TERMINATED si la búsqueda fue cancelada.

Para realizar una búsqueda de servicios también usaremos la clase DiscoveryAgent e implementaremos la interfaz DiscoveryListener. En este caso nos interesarán los métodos servicesDiscovered() y serviceSearchCompleted() de la interfaz DiscoveryListener. Para comenzar la búsqueda usaremos searchServices() de la clase DiscoveryAgent. Este método es un poco complejo así que lo vamos a ver con más detalle:

public int searchServices(int[] attrSet, UUID[] uuidSet, RemoteDevice btDev, DiscoveryListener discListener) throws BluetoothStateException;

El primer argumento es un array de enteros con el que especificaremos los atributos de servicio en los que estamos interesados. Como veremos más adelante, cuando hablemos de la interfaz ServiceRecord, los servicios son descritos a través de atributos que son identificados numéricamente. Pues bien, este array contendrá los identificadores de estos atributos. El segundo argumento es un array de identificadores de servicio. Nos permite especificar los servicios en los que estamos interesados. El tercer argumento es el dispositivo remoto sobre el que vamos a realizar la búsqueda. Por último argumento pasaremos un objeto que implemente DiscoveryListener, que será usado para notificar los eventos de búsqueda de servicios.

Es posible que queramos hacer diversas búsquedas de servicios al mismo tiempo. El método searchServices() nos devolverá un entero que identificará la búsqueda. Este valor entero nos servirá para saber a qué búsqueda pertenecen los eventos servicesDiscovered() y serviceSearchCompleted(). Comenzemos a ver los métodos:

-public void serviceSearchCompleted(int transID, int respCode)

Este método es llamado cuando se finaliza un proceso de búsqueda. El primer argumento identifica el proceso de búsqueda (que recordemos que es el valor devuelto al invocar el método searchServices() de la clase DiscoveryAgent). El segundo argumento indica el motivo de finalización de la búsqueda.

Podemos cancelar un proceso de búsqueda de servicios llamando al método cancel-ServiceSearch() pasándole como argumento el identificador de proceso de búsqueda, que es el número entero devuelto cuando se comenzó la búsqueda con searchServices().

-public void servicesDiscovered(int transID, ServiceRecord[] sr)

Este método nos notifica que se han encontrado servicios. El primer argumento es un entero que es el que identifica el proceso de búsqueda. Este entero es el mismo que devolvió searchDevices() cuando se comenzó la búsqueda. El segundo argumento es un array de objetos ServiceRecord.

Un objeto ServiceRecord describe las características de un servicio bluetooth. Un servicio Bluetooth se describe mediante atributos, los cuales se identifican numéricamente, es decir, un servicio Bluetooth tiene una lista de pares identificador-valor que lo describen. El objeto ServiceRecord se encarga de almacenar estos pares. Los identificadores son números enteros y los valores son objetos de tipo DataElement.

-La clase DataElement

La clase DataElement se encarga de encapsular los tipos de datos disponibles para describir un atributo de servicio. Estos tipos de datos son: valor nulo, enteros de diferente longitud, arrays de bytes, URLs, UUIDs, booleanos, Strings o enumeraciones (java.util.Enumeration) de los tipos anteriores.

Para saber qué tipo de dato está almacenando un DataElement usaremos el método getDataType(). Según el tipo de dato que esté almacenando usaremos un método diferente para obtenerlo. Si está almacenano un valor booleano usaremos el método get-Boolean(), si se trata de un entero usaremos el método getLong() y en otro caso usaremos el método getValue() que devolverá un java.lang.String si está almacenando una URL o un String, o bien devolverá un javax.bluetooth.UUID si se trata de un UUID, o bien un java.util.Enumeration.

Comunicación

El API javax.bluetooth permite usar dos mecanismos de conexión: SPP y L2CAP. Mediante SPP obtendremos un InputStream y un OutputStream. Mediante L2CAP enviaremos y recibiremos arrays de bytes.

Para abrir cualquier tipo de conexión haremos uso de la clase javax.microedition.io.Connector. En concreto usaremos el método estático open() que está sobrecargado; su versión más sencilla requiere un parámetro que es un String que contendrá la URL con los datos necesarios para realizar la conexión. La URL será diferente dependiendo si queremos ser cliente o servidor de una conexión L2CAP o SPP.

El MIDlet de nuestro proyecto utiliza una conexión SPP basada en InputStream, ya que la comunicación entre el teléfono móvil y la estación base consiste principalmente en un una recepción de datos nuevos para el cliente (teléfono).

Comunicación cliente

Obtendremos la URL necesaria para realizar la conexión a través del método getConnectionURL() de un objeto ServiceRecord. Recordemos que un objeto ServiceRecord representa un servicio, es decir una vez hayamos encontrado el servicio deseado (un objeto ServiceRecord), él mismo nos proveerá la URL necesaria para conectarnos a él.

Comunicación cliente SPP

Una vez tenemos la URL con la información necesaria usaremos el método Connector.open() para realizar la conexión. Este método devuelve un objeto distinto según el tipo de protocolo usado. En nuestro caso, el de un cliente SPP, devolverá un StreamConnection.

A partir del StreamConnection podremos obtener los flujos de entrada y de salida:

```
StreamConnection con = (StreamConnection) Connector.open(url);
OutputStream out = con.openOutputStream();
InputStream in = con.openInputStream();
```

5.4 Entorno de desarrollo integrado Eclipse

El conjunto de paquetes, entornos y herramientas de desarrollo y programación utilizado en este proyecto está formado por el entorno Eclipse + Eclipse ME, junto con el SDK de Nokia 6131 NFC más la herramienta de edición, compilación y ejecución Carbide.j, también de Nokia.

5.4.1 Eclipse

Eclipse es principalmente una plataforma de programación, usada para crear entornos integrados de desarrollo (del Inglés IDE).

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto.

Facilita enormemente las tareas de edición, compilación y ejecución de programas durante su fase de desarrollo. Aunque Eclipse pretende ser un entorno versátil soportando varios lenguajes de programación, es con el lenguaje Java con el que mejor se integra y con el que ha ganado su popularidad.

Eclipse es una aplicación gratuita, disponible en la red para su descarga e incluida ya en muchas distribuciones de Linux. Eclipse proporciona el entorno de desarrollo solamente, siendo necesario además, para el caso del lenguaje Java, disponer del Java Development Kit o JDK para poder compilar y ejecutar las aplicaciones desarrolladas.

5.4.2 Eclipse ME

Eclipse ME es un plugin que se añade a Eclipse para el desarrollo de aplicaciones en J2ME, como es el caso de nuestro proyecto. Gracias a este plugin, podemos disfrutar de las ventajas de este IDE: autocompletado, resaltado de sintaxis, sencillez de programación y mejor disponibilidad espacial para crear aplicaciones J2ME. Sin este plugin , la programación de J2ME en Eclipse seria sin las ayudas anteriormente descritas y habría que cargar el código con el módulo KToolbar del Sun Wíreles Toolkit. Para cargar el código en nuestro teléfono móvil de Nokia dispondremos de la herramienta Carbide y del propio SDK de Nokia 6131 NFC.

Para instalar el plugin deberemos buscar nuevas actualizaciones para nuestro IDE de Eclipse, tarea que se hace en el menú Help/Software Updates/Find and Install:

Seleccionaremos la opción “Search new features to install” y daremos a siguiente. Nos aparecerá una ventana con todos los sitios remotos en los que se buscarán actualizaciones. El sitio del plugin no estará seleccionado por defecto así que tendremos que añadirlo pinchando en “new Remote Site” y completando la ventana que se abrirá del siguiente modo:

Una vez hecho esto comprobaremos que el sitio se ha añadido correctamente si en la ventana de sitios de actualizaciones aparece el que acabamos de crear. Pincharemos en Finish para iniciar la búsqueda y posteriormente seleccionaremos todas las actualizaciones disponibles. Pulsaremos en siguiente y aceptaremos los términos de la licencia para comenzar la descarga. Una vez finalizada la descarga pulsaremos el botón siguiente (ext) para confirmar los términos de la licencia para la instalación del plugin. En este punto pulsaremos en el botón ext, y aparecerá en pantalla información detallada del plugin a instalar y el directorio de instalación. Realizaremos la instalación pulsando el botón “install all”.

5.4.3 Carbide.j 1.5

Carbide.j es una herramienta para el desarrollo de software con J2ME que mejora el diseño y la verificación de las aplicaciones para los dispositivos de Nokia. Suministra herramientas para crear aplicaciones Mobile Information Device Profile (MIDP) y Personal Profile (PP). Permite a los desarrolladores obtener el máximo provecho de las prestaciones que ofrece la tecnología Java ME. A lo largo de su instalación existe un momento en el cual se pregunta al usuario con en qué IDE desea incorporar las funciones de Carbide.j, en nuestro caso Eclipse.

Carbide.j suministra principalmente herramientas para crear clases y paquetes de aplicación, emular y desplegar MIDlets. Las herramientas son cargadas al inicio de Eclipse automáticamente.

5.4.4 Nokia 6131 NFC SDK

El SDK de NFC de Nokia 6131 permite que programadores creen y emulen aplicaciones de Java (MIDlets) para el teléfono móvil Nokia 6131 NFC.

Incluye la API de comunicación Contactless (JSR-257), que permite el uso de las características de comunicación de campo cercano (NFC). Además el SDK incluye extensiones para otras conexiones y configuraciones como la API JSR-82 para Bluetooth.

Otra característica muy importante es que contiene un emulador de teléfono de Nokia 6131 NFC, muy útil para comprobar exactamente el modo de funcionamiento y la manera en que se mostrarán las pantallas del MIDlet que se desarrolle con Eclipse ME y Carbide.j en el teléfono físico.

5.4.5 Desarrollo y despliegue de MIDlets

A continuación se describen los principales pasos que se siguieron para la creación y despliegue (deployment) del Midlet principal de nuestra aplicación Bluetooth en el teléfono móvil Nokia 6131 NFC, sin entrar en detalle en el código java ME de las clases que forman el proyecto. Esta descripción es un ejemplo de los pasos a seguir para desarrollar cualquier Midlet en Eclipse para nuestro dispositivo.

- Como utilizamos el Nokia Developer Suite para J2ME, en las opciones seleccionamos Java MIDP Project (Nokia SDK Plug-in), como se aprecia en la figura 1.

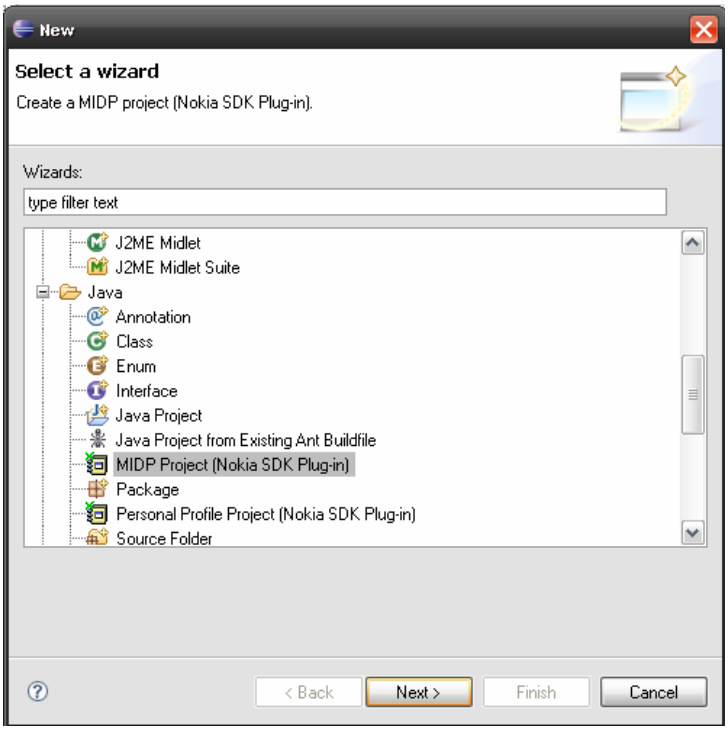


Figura 1

- Se asigna el nombre al proyecto (APLICACIÓN BLUETOOTH FINAL) y a continuación seleccionamos el SDK (emulador) con el cual se desea trabajar. Elegimos el de Nokia 6131 NFC previamente instalado.

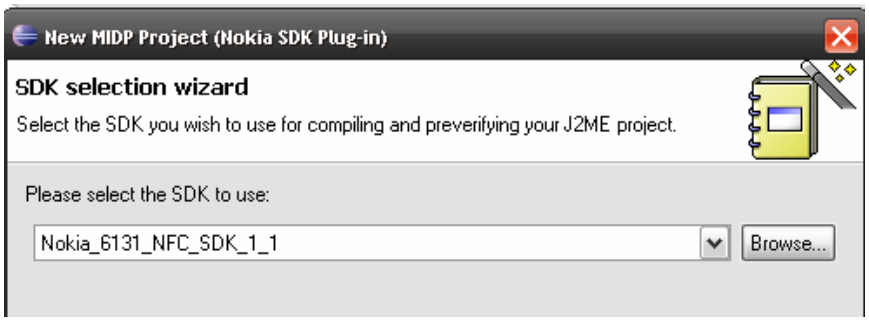


Figura 2

- Con esto se finaliza el proceso de creación del proyecto. Asegurándonos de que se crean los directorios para el código fuente (src) y para el código compilado (bin)

- La barra de herramientas más importante es la formada por los botones de Carbide.j de Nokia, herramienta para el desarrollo de aplicaciones en J2ME bajo Eclipse (ver Figura 3)

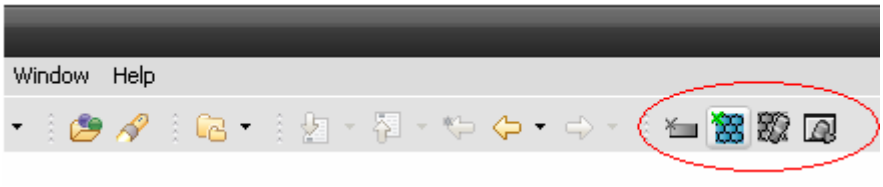


Figura 3

Los cuatro botones permiten en su orden: Crear una nueva clase MIDlet, crear un nuevo paquete de aplicación J2ME, implantar o desplegar (deployment) la aplicación y emular la aplicación.

- Una vez creadas todas las clases que formen el proyecto y compilar este sin errores, para poder ejecutar el Midlet creado, ya sea en el emulador o en el propio dispositivo, se debe empaquetar la aplicación, es decir, crear los archivos jar y jad. Esto se hace presionando el segundo botón de la barra de herramientas anteriormente descrita y clickando en “Generate” (ver Figura 4)

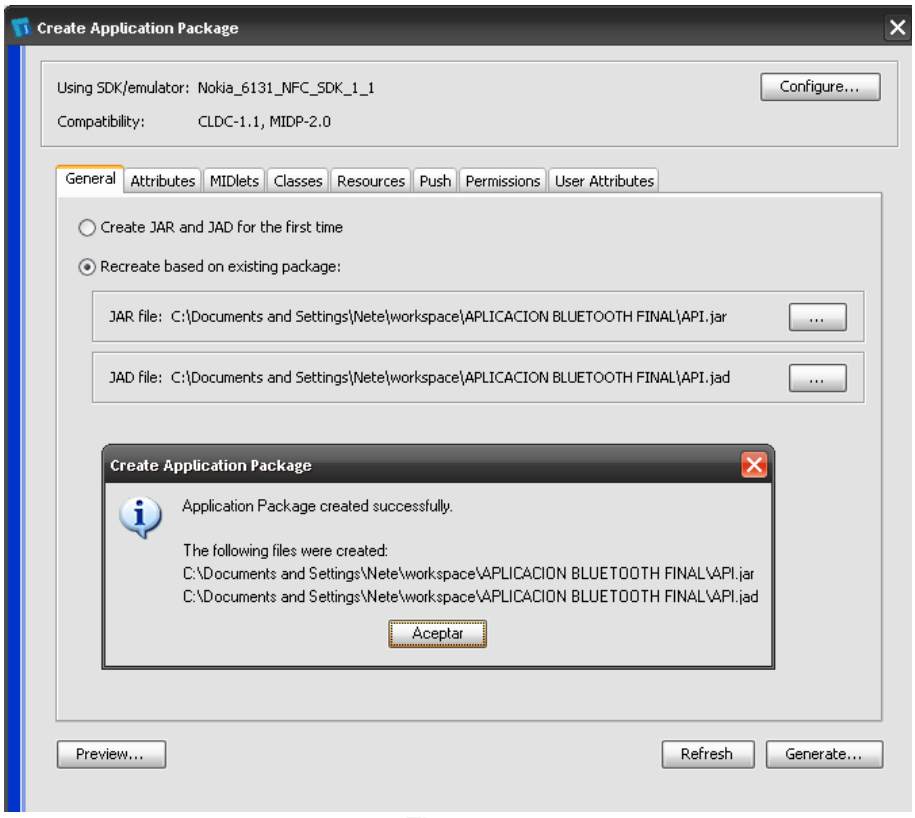
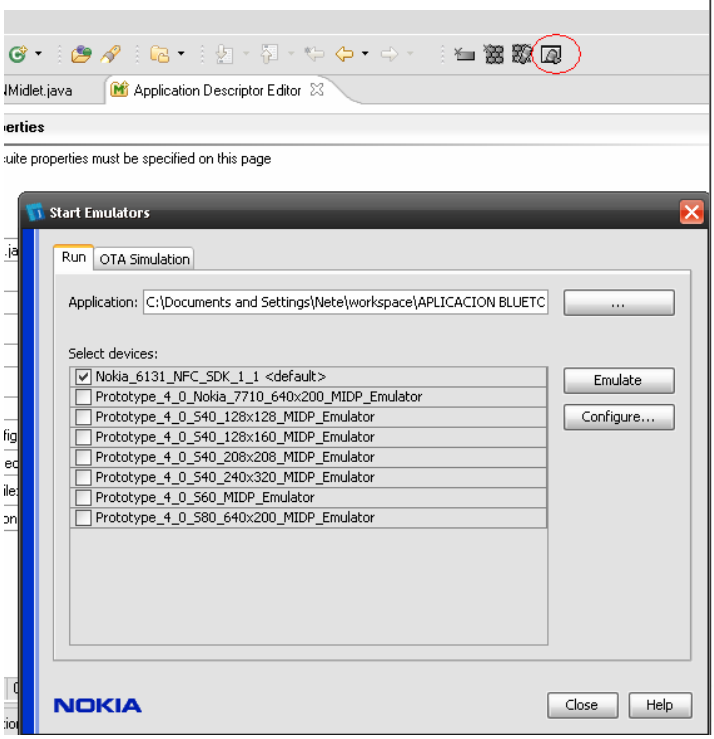


Figura 4

- Ya se podría proceder al “deployment” en el teléfono móvil con ayuda del tercer botón o, si se prefiere, ejecutar y preverificar la aplicación en el simulador que proporciona el SDK de Nokia 6131 NFC. Recordar que accedemos al emulador pulsando en el cuarto botón de la barra de herramientas que nos proporcionó la instalación del paquete Carbide.j(ver Figura 5). Podemos ver una captura del inicio de la emulación en la figura 6.



(Figura 5)

(Figura 6)

Mariano Gallardo Magaña
Ingeniería en Automática y Electrónica Industrial

5.5 Aplicación (MIDlet) de monitorización

El título del MIDlet es el de: “Sistema de monitorización Bluetooth de los contadores de consumo eléctrico del puerto de Génova”

5.5.1 Descripción general del Midlet

El Midlet está formado por una biblioteca de clases que se encarga, por un lado, de conectar el teléfono vía Bluetooth con un dispositivo externo que lo abastece de nueva información de los grupos de contadores y por otro lado de gestionar el acceso y manipulación de los registros que se van almacenando en nuestro dispositivo móvil, incluyendo funcionalidades de visualización, carga, creación, búsqueda y borrado de registros. Las interfaces están basadas en la arquitectura J2ME y el subconjunto de APIs asociadas. Se plantea una navegación con dos funcionalidades de entrada:

-Carga Bluetooth
-Monitorización y manipulación de registros. Ver figuras 1 y 2

Con la carga Bluetooth el teléfono móvil será capaz de buscar dispositivos cercanos con conectividad Bluetooth, conocer si están ofreciendo comunicación serie y conectarse y recibir nuevos datos en forma de registros de información de contadores.

Si se opta por la funcionalidad llamada “Menú Aplicación” accederemos a la parte del Midlet donde el usuario podrá realizar consultas y gestiones sobre la información almacenada de los Grupos Contadores. A partir de la búsqueda de registros, sobre el listado de resultados obtenidos, se podrán realizar las consultas de consumos eléctricos, borrados o visualización en detalle de los registros.

(Fig.1)

(Fig.2)

5.5.2 Carga Bluetooth

Es la funcionalidad más importante del MIDlet, encargada de la conexión y la recepción de datos vía Bluetooth, al ser la principal de la aplicación se ha desarrollado en la clase principal del programa, llamada BluetoothWSNMidlet, donde se encuentra el inicio, la pausa y la destrucción del Midlet (starApp, pauseApp y destroyApp). Desde esta clase también accederemos al resto de ventanas y formularios que componen el MIDlet.

Ejecución de la carga Bluetooth:

La primera ventana que aparece nada más seleccionar la opción de Carga de datos Bluetooth desde el menú inicial nos informa de que el teléfono móvil esta buscando dispositivos cercanos que ofrezcan datos a través de comunicación inalámbrica Bluetooth (ver Fig.3).

(Fig.3)

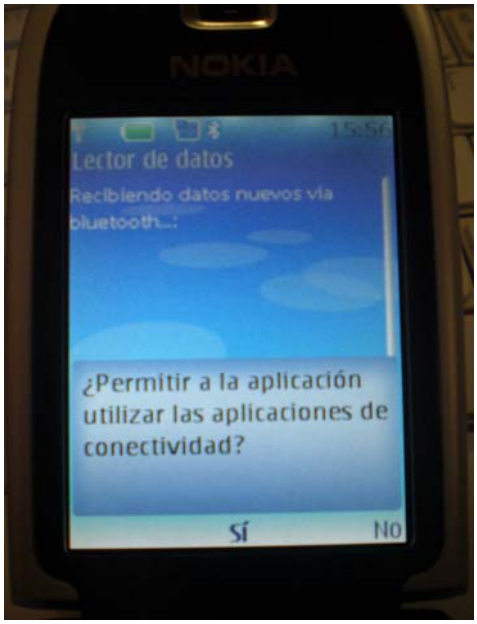
Una vez detectados los dispositivos con conectividad Bluetooth cercanos, se procede a la selección del indicado para la recepción de datos Bluetooth. En nuestro sistema en concreto, el dispositivo que abastecerá al teléfono móvil de nuevos datos se llama “DANIEL” (ver Fig.4).

(Fig.4)

El siguiente texto que aparece en pantalla informa si el dispositivo seleccionado ofrece comunicación por puerto serie y si está listo para realizar comunicación. Además informa de datos de la conexión y la dirección URL del dispositivo encontrado y con el que se va a realizar la conexión (cada dispositivo Bluetooth tiene una dirección asignada de 48 bits) (ver Fig.5).

(Fig.5)

Después de proceder eligiendo el comando de “Conectar” el MIDlet pregunta al usuario si permite conectividad para la recepción de datos externos (Fig. 6)



(Fig.6)

Si se acepta se iniciará la recepción de datos. La duración será de unos segundos, dependiendo de la longitud de la cadena de entrada de datos. La cadena de entrada de datos está compuesta de 3 tramos principales. Uno que contiene información sobre la temperatura de la Estación Base y fuente de datos, otro que informa del nivel de batería de la Estación Bluetooth y por último, la zona de la cadena que contiene los registros del grupo de contadores eléctricos. La longitud de la cadena dependerá del nº de registros que se ofrezcan y sean recibidos en cada conexión. Un ejemplo de cadena sería el siguiente:

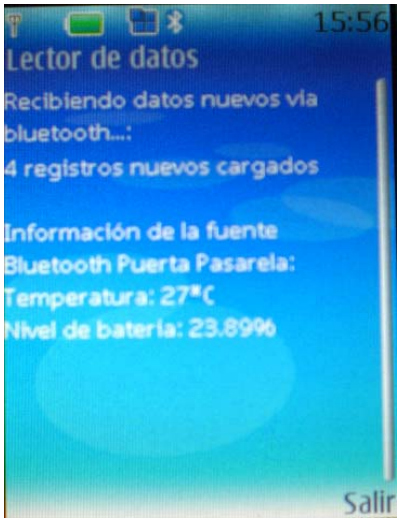
Ejemplo:

27ºC23.89%-1:12#12#2008:10*00-2:11#04#2008:07*10:-2:13#04#2008:09*40:-3:21#04#2008:11*30:

Donde serían cargados 4 nuevos registros.

En el anexo del proyecto, donde se detalla el código fuente de la clase BluetoothWSNMidlet, puede verse como se realiza el tratamiento y la extracción de información de esta cadena de entrada de datos.

En la pantalla de lectura de datos del teléfono móvil aparecerán:
-Nº de registros nuevos cargados
-Temperatura
-Nivel de batería



(Fig.7)

5.5.3 Gestión de registros (RMS)

Nuestro dispositivo móvil Nokia 6131 NFC no dispone de disco duro donde almacenar información permanentemente. Mediante el RMS (*Record Management System*) resolveremos el problema. RMS es un pequeño sistema de bases de datos muy sencillo, pero que nos permite añadir información en una memoria no volátil del móvil.

En una base de datos RMS, el elemento básico es el registro (*record*). Un registro es la unidad de información más pequeña que puede ser almacenada. Los registros son almacenados en un recordStore que puede visualizarse como una colección de registros.

Cuando almacenamos un registro en el recordStore, a éste se le asigna un identificador único que identifica unívocamente al registro. La capacidad máxima de memoria asignada para el recordStore en el Nokia 6131 NFC es de 262Kb. Por lo tanto, si cada registro esta formado por 19 bytes, la cantidad máxima de registros que podrá albergar el recordStore será del orden de 13800 registros.

Al cada registro le acompañará un identificador de registro.

La clase BDatos contiene un repositorio de registros que creamos al instanciarla y todas las operaciones asociadas al uso de los mismos. Las operaciones que nos permite esta clase son:

- Generar un repositorio de registros a gestionar
- Abrir un repositorio existente para gestionarlo
- Cerrar un repositorio gestionado
- Borrar un repositorio gestionado
- Insertar registros en el repositorio
- Eliminar registros del repositorio
- Buscar identificadores de registros asociados a unos parámetros de búsqueda

- Recuperar los registros con identificadores dados
- Eliminar registros con unos identificadores dados
- Definir un filtro de repositorio
- Eliminar el filtro asociado a un repositorio
- Recuperar el valor de una columna del registro

5.5.4 Registros

En nuestra aplicación de monitorización en cuestión, cada registro representa la lectura de ¼ de KW/h por parte de algún Grupo Contador en particular. El registro contiene información del nº de Grupo de Contador, el día y la hora en que se produjo esa lectura de pulso. El formato es el siguiente:

Nº Grupo Contador:Dia:Hora:

Es una cadena (java.lang.String) que va separada por el carácter definido en la variable de clase SEPARATOR. En la actualidad este carácter es ":".

Ejemplo-> 1:12#03#2008:22*30:

Cada uno de los campos del registro se tratará como una *columna*, comenzando en la posición 0.

Nº	Grupo Contador
1	San Desiderio
2	San Lorenzo
3	San Giobatta

El uso de los caracteres # y * resulta más cómodo y rápido a la hora de insertar un registro nuevo manualmente, o a la hora de solicitar el listado de registros de lecturas para un día en concreto o de un Grupo Contador en particular. Se lo hacemos notar al programa con la restricción *PHONENUMBER* de la subclase *Textlist* que especifica que el campo debe contener los caracteres típicos de un teclado de teléfono móvil. La subclase *Textlist* es la que nos permite editar un campo de texto (normalmente de una línea) dentro de un *Form* (que contiene los ítems del interfaz gráfico).

5.5.5 Filtro

Un repositorio gestionado por BDatos poseerá un filtro. El filtro dará cómo válidos sólo los registros que cumplan la condición de tener un valor concreto en una columna dada del registro. Si se tuviesen que cumplir dos condiciones para satisfacer la consulta, en el caso de buscar los registros ocurridos en un día en concreto y para un Grupo de Contador en particular, se utiliza dos veces el filtro de la BDatos y se buscan que registros cumplen las dos condiciones por separado. Esto se puede ver en detalle observando el código fuente que forma las clases de "ListadoResultado" y "ListadoResultadoble" del anexo.

Ejemplo:

Dados los registros:

1:12#02#2008:10*00:
2:14#04#2009:11*30:
1:12#02#2008:12*00:

Si realizamos una consulta que solicite los pulsos registrados en el Grupo Contador 2 (San Lorenzo), es decir, que indique que la columna 1 tiene que tener el valor “2”, utilizando la clase “ListadoResultado” sólo se recuperará:

2:14#04#2009:11*30:
(Una consulta de filtro)

Si quisiésemos saber los pulsos de ¼ de KW/h registrados por el Grupo contador 1 (San Desiderio) y en el día 12 de Febrero de 2008, con la clase “ListadoResultadoble” obtendríamos:

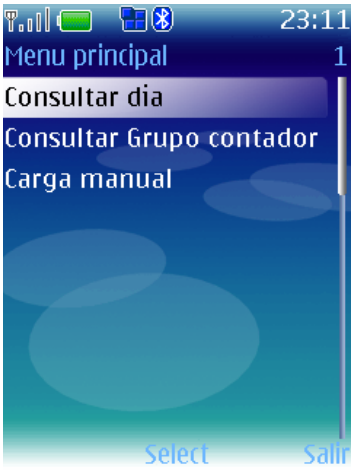
1:12#02#2008:10*00:
1:12#02#2008:12*00:
(Doble consulta de filtro)

Las instancias de BDatos pueden no tener filtros, con lo que se podrán recuperar todos los registros que contengan o bien tener un único filtro activo. El filtro se puede cambiar tantas veces como sea necesario.
Cuando se crea el filtro se proporciona el valor de la columna por la que se filtrará y el valor que se utilizará en la comparación.

5.5.6 Menú Principal

Es el menú principal para realizar las consultas y hacer gestiones sobre la base de datos que contiene los registros de información de consumo de los contadores. A partir del menú principal accedemos a las pantallas de la interfaz de consultas e inserción manual de registros.
Son 3:

- Consulta de consumo diario
- Consulta según Grupo Contador
- Carga de datos Manual



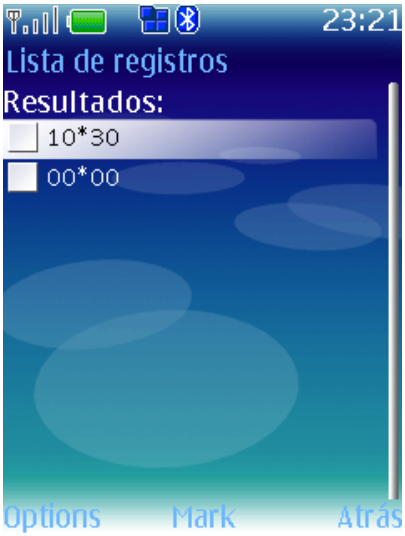
(Fig.8)

5.5.7 Ventana de consulta de consumo diario

Desde la ventana de consulta de consumo diario el usuario podrá escribir el día para el que desea realizar la consulta de consumo eléctrico y seleccionar la búsqueda de manera que se obtengan los registros de un determinado Grupo Contador o de todos los grupos para ese día (ver Fig.9).

(Fig.9)

Se lanzará una consulta contra el listado de registros que nos devolverá todos los identificadores de registro que verifiquen la consulta realizada. Si la consulta ha realizado dos filtros, es decir, uno para el día y otro para un grupo contador en concreto, la información más significativa de los registros filtrados que aparecerá en la ventana de listado de resultados serán las horas en las que se produjeron los pulsos de contador en ese grupo contador y día en particular (ver Fig.10).



(Fig.10)

Si la consulta diaria se realiza para todos los grupos de contadores, se utilizará sólo un filtro, para el día. El listado de resultados estará formado por los números que identifican a cada Grupo Contador de cada registro encontrado en la consulta (ver Fig.11).

(Fig.11)

Si no existen datos para la consulta realizada, se mostrará una ventana de error indicándolo (ver Fig.12)

(Fig.12)

5.5.8 Ventana de consulta según Grupo Contador

Desde esta ventana de consulta podremos consultar el historial de consumo de cada Grupo Contador. Incluyéndose también la posibilidad de realizar una consulta del total de los registros almacenados en la base de datos (ver Fig.13).

(Fig.13)

El listado de resultados mostrará las fechas en que se produjeron las lecturas, considerándose la información más significativa de los registros filtrados según nº de Grupo Contador (ver Fig.14)

(Fig.14)

Al igual que sucede en todas las ventanas de listado de resultado se podrá ver en detalle cada registro.

5.5.9 Ventana de carga de datos manual

Desde este formulario del MIDlet podremos introducir la información que forma un registro a través del teclado del teléfono móvil, es decir, manualmente. Esta posibilidad manual de inserción de datos es sólo para realizar pruebas en la base de datos ya que los registros se adquieren vía Bluetooth normalmente. El formulario de carga manual nos pide:

-nº Grupo Contador
-Día
-Hora

(Fig.15)

Se define un método, *validateFields ()*, que se podrá emplear para certificar que toda la información introducida en la ventana es correcta. Si nos olvidáramos de introducir alguno de los campos una pantalla de alerta de error nos avisaría de que debemos introducir todos los campos (ver Fig.16).

(Fig.16)

5.5.10 Ventana de listado de resultados

La ventana de listado de resultados cumple dos funciones:

- Mostrar los resultados que cumplen el filtrado de datos realizado en cualquiera de las dos ventanas de consultas

- Servir como enlace con las operaciones de ver detalle, consumo eléctrico y borrado de algún o alguno de los registros seleccionados

Si deseamos "Ver Detalle", sólo podremos seleccionar un elemento. Si intentamos seleccionar más de uno, nos indicará que esa operación no se puede realizar (ver Fig.17).

(Fig.17)

Al seleccionar la operación de consulta de consumo eléctrico, una nueva ventana nos mostrará la cantidad de KW/h consumidos según la petición de consulta seleccionada. El consumo se calcula según la cantidad de registros que forman el listado de resultados (ver Fig.18).

(Fig.18)

Ejemplo:

Si el listado de resultados lo forman 8 registros, al elegir consumo, sería indiferente si alguno en particular estuviese marcado, será de 2KW/h, ya que cada registro indicaba la lectura de ¼ de KW/h por alguno de los grupos contadores.

Si se selecciona borrar uno o más registros, se solicitará confirmación y se procederá al borrado de los mismos si el usuario lo confirma.

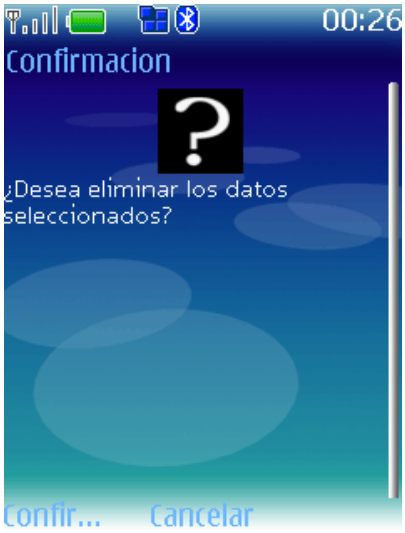
5.5.11 Ventana de detalle de registros

Desde esta ventana se puede consultar el detalle de un registro seleccionado en la ventana de listado de resultados (ver Fig.19).

(Fig.19)

5.5.12 Ventana de borrado de registro

La ventana de borrado es una ventana de confirmación de eliminación de datos más que una interfaz con contenidos de registro propiamente dicha (ver Fig.20).



(Fig.20)

Una vez eliminados los registros, retornamos a la ventana de lista de registros donde se actualiza la información.

5.5.13 Ventana Alerta Confirmación

Esta ventana se utiliza para mostrar mensajes de alerta de confirmación para la notificación de la correcta terminación de alguna opción del sistema. No incorpora ninguna funcionalidad interactiva y sólo se limita a mostrar un mensaje. Ejemplo, cuando introducimos un registro de forma manual correctamente:

(Fig.21)

5.5.14 Ventana Alerta Error

Esta ventana se utiliza para mostrar mensajes de error e indicar la imposibilidad de ejecutar alguna operación del sistema. No incorpora ninguna funcionalidad interactiva y sólo se limita a mostrar un mensaje. Ver Fig.16 y Fig.17 como ejemplo.

5.5.15 Cuadro de Diálogo

Ventana que presenta una pregunta de forma interactiva para la que se puede contestar y procesar las respuestas dadas y actuar en consecuencia.

Es una construcción que permite responder a las preguntas de tipo binario “SI-NO” o “CONFIRMAR-CANCELAR”. Aparece por ejemplo en la pantalla de borrado de registro (ver Fig.20).

6 CONCLUSIONES Y FUTUROS DESARROLLOS

El objetivo principal de este proyecto ha sido el de desarrollar un MIDlet de monitorización para una red de sensores inalámbrica.

Sin embargo, no ha sido el único objetivo. Para la elaboración de un Proyecto Fin de Carrera es necesario llevar a cabo un proceso de estudio, análisis y diseño sobre la idea que se pretende desarrollar. Por lo tanto, tras la propuesta sobre la creación de un sistema de monitorización a implantar en un dispositivo móvil, se procedió a realizar una búsqueda y elección dentro de varios posibles entornos de desarrollo, teléfonos móviles, tecnologías de comunicación inalámbrica y demás necesidades que pudieran estar involucradas en la creación del proyecto. Además ha sido necesario el aprendizaje del lenguaje de programación Java ME para el desarrollo de MIDlets en dispositivos móviles.

Se ha podido comprobar gracias a la red la cantidad de estudiantes, profesores, usuarios e investigadores volcados e inmersos en el mundo de las comunicaciones inalámbricas y de los usos y aplicaciones que permiten desarrollar.

En cuanto al funcionamiento del software creado en este proyecto, se pueden evaluar dos aspectos principalmente, uno es el de la comunicación que realiza el MIDlet con la estación base Bluetooth y otro el del tratamiento y gestión de datos por parte de la aplicación dentro del teléfono móvil. En ambos aspectos el funcionamiento ha sido el correcto y el esperado.

Futuros desarrollos

• Transmitir la información recopilada por el teléfono a través de SMS y/o MMS a otra posible estación situada fuera del alcance Bluetooth.

• Extender el campo de contadores a monitorizar, requiriéndose para ello más estaciones y la creación de protocolos de comunicación entre estas y el dispositivo que se encargue de la recepción y gestión de datos.

• Realizar el mismo proyecto pero sustituyendo la estación base bluetooth por un servidor web y utilizando dispositivos móviles con conexión a internet para monitorizar los resultados.

• Darle a esta aplicación una mayor funcionalidad, añadiendo el control de otro tipo de actuadores, tales como alarmas de temperatura o alarmas de máximos y mínimos de consumo.

• Adaptar, mejorar y/o añadir funciones al MIDlet para otro sistema operativo o plataforma de telefonía móvil, como es el caso de Symbian o Maemo

7 APÉNDICE

7.1 Código Fuente

A continuación se muestra el contenido de los archivos .java que forman la aplicación de monitorización.

7.1.1 BluetoothWSNMidlet

```
package wsn;

import java.util.*;
import java.io.*;
import javax.microedition.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import wsn.MenuPrincipal;
import wsn.BluetoothWSNMidlet;

public class BluetoothWSNMidlet extends MIDlet implements CommandListener{


static BDatos vBD = new BDatos("lecturas");
private MenuPrincipal vMenuPrincipal = new MenuPrincipal();

static BluetoothWSNMidlet instance;
public Display display;
public Form discoveryForm;
public Form consultaForm;
public Form InicioForm;
public Form readyToConnectForm;
public Form dataViewForm;
public ImageItem mainImageItem;
public Image mainImage;
public ImageItem iniItem;
public Image ini;
public Image bt_logo;
public TextField addressTextField;
public TextField subjectTextField;
public TextField messageTextField;
public Command selectCommand;
public Command bluetooth;
public Command tk;
public Command menu;
public Command exitCommand;     /*del discovery*/
public Command exitCommandp;   /*del principal*/
public Command connectCommand;
public List devicesList;
public Thread btUtility;
public String btConnectionURL;
public StreamConnection connection;
public InputStream in;
public boolean readData = false;


public BluetoothWSNMidlet() {

       BluetoothWSNMidlet.instance = this;
```

```
display=Display.getDisplay(this);
InicioForm= new Form("Monitorización Bluetooth");
try{
	ini = Image.createImage(Image.createImage("/modify.png"));

} catch (java.io.IOException e){
	e.printStackTrace();
}

iniItem = new ImageItem("SMBC", ini, Item.LAYOUT_CENTER, "");
InicioForm.append(iniItem);
InicioForm.append("\nSistema de monitorización Bluetooth de los
	contadores de consumo eléctrico del puerto de Génova  .\n\n");

exitCommandp = new Command("Salir", Command.EXIT, 1);
tk = new Command( "<---", Command.SCREEN,1);
menu = new Command("Menu Aplicación ", Command.SCREEN, 1);
bluetooth=new Command("Carga datos Bluetooth",Command.SCREEN,1);

InicioForm.addCommand(exitCommandp);
InicioForm.addCommand(tk);
InicioForm.addCommand(bluetooth);
InicioForm.addCommand(menu);
InicioForm.setCommandListener(this);


////Consulta
consultaForm=new Form ("Consulta consumo");


////DiscoveryForm.Bluetooth inicio

discoveryForm = new Form("Bluetooth WSN");

	try{
		mainImage = Image.createImage("/loading.png");
		bt_logo = Image.createImage("/loading.png");
	} catch (java.io.IOException e){
		e.printStackTrace();
	}
mainImageItem = new ImageItem("Bluetooth WSN", mainImage,
Item.LAYOUT_CENTER, "");

discoveryForm.append(mainImageItem);
discoveryForm.append("\nSe escaneará el área buscando dispositivos
bluetooth y determinará si alguno ofrece servicios de comunicacion
inalámbrica por puerto serie.\n\n");


exitCommand = new Command("Salir", Command.EXIT, 1);
discoveryForm.addCommand(exitCommand);
discoveryForm.setCommandListener(this);

/// devicesList 1

devicesList = new List("Selecciona un dispositivo bluetooth",
	Choice.IMPLICIT, new String[0], new Image[0]);

selectCommand = new Command("Seleccionar", Command.ITEM, 1);
devicesList.addCommand(selectCommand);
devicesList.setCommandListener(this);
devicesList.setSelectedFlags(new boolean[0]);

	/// readyToConnectForm
```

```
readyToConnectForm = new Form("Listo para conectar");
readyToConnectForm.append("El dispositivo bluetooth seleccionado está
ofreciendo un servicio válido por puerto serie, y está preparado para
conectar. Presionar 'conectar'para conectar y leer los datos.");

connectCommand = new Command("Conectar", Command.ITEM, 1);
readyToConnectForm.addCommand(connectCommand);
readyToConnectForm.setCommandListener(this);

       /// inicializacion del dataViewForm

dataViewForm = new Form("Lector de datos");
dataViewForm.append("Recibiendo datos nuevos via bluetooth...:\n\n");
dataViewForm.addCommand(exitCommand);
dataViewForm.setCommandListener(this);

}


public void commandAction(Command command, Displayable d) {


       if(command == selectCommand) {
              btUtility.start();
       }
       if(command == exitCommand ) {
              display.setCurrent(InicioForm);
       }
       if(command == exitCommandp ) {
              readData = false;
              destroyApp(true);
       }
       if(command == connectCommand ) {
              Thread commReaderThread = new COMMReader();
              commReaderThread.start();
              display.setCurrent(dataViewForm);
       }
       if(command == bluetooth ) {

              display.setCurrent(discoveryForm);
              btUtility = new BTUtility();
    }
       if(command == menu ) {

       Display.getDisplay(this).setCurrent(vMenuPrincipal);


    }
        if(command == tk ) {

              //para meter los comandos del menu en el apartado de
                     opciones
    }
}


public void startApp() {
       display.setCurrent(InicioForm);
}


public void pauseApp() {
```

```
        }

        public void destroyApp(boolean b) {
                vBD.closeStore();
                notifyDestroyed();
        }


///////////////

    /**
     * A continuación la 'clase' que se encarga de descubrir dispositivos
bluetooh
     */
    class BTUtility extends Thread implements DiscoveryListener {

        Vector remoteDevices = new Vector();
        Vector deviceNames = new Vector();

        DiscoveryAgent discoveryAgent;

        //  0x1101 es la UUID para
        // el perfil Serie
        UUID[] uuidSet = {new UUID(0x1101) };

        // 0x0100 es el atributo para el elemento nombre de servicio
        // en el 'service record'
        int[] attrSet = {0x0100};


        public BTUtility() {
            try {
                LocalDevice localDevice = LocalDevice.getLocalDevice();
                discoveryAgent = localDevice.getDiscoveryAgent();
                discoveryForm.append(" Buscando dispositivos bluetooth ..\n");
                discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this);

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

        public void deviceDiscovered(RemoteDevice remoteDevice, DeviceClass
             cod) {
            try{
                discoveryForm.append("Encontrado:\n " +
                            remoteDevice.getFriendlyName(true)+"\n");
            } catch(Exception e){
                discoveryForm.append("Encontrado:\n " +
                            remoteDevice.getBluetoothAddress()+"\n");
            } finally{
                remoteDevices.addElement(remoteDevice);
             }
        }


        public void inquiryCompleted(int discType) {

        if (remoteDevices.size() > 0) {

                // El proceso de búsqueda fue satisfactorio
                // y muestra una lista con los dispositivos encontrados al
                          usuario
```

```
for (int i=0; i<remoteDevices.size(); i++){
     try{

       devicesList.append(((RemoteDevice)remoteDevices.elementAt(i)).
 getFriendlyName(true), bt_logo);

       } catch (Exception e){

 devicesList.append(((RemoteDevice)remoteDevices.elementAt(i)).getBluetoo
       thAddress(), bt_logo);
                                          }
          }
display.setCurrent(devicesList);
      } else {

       }

  }

  public void run(){

      try {
          RemoteDevice remoteDevice =
 (RemoteDevice)remoteDevices.elementAt(devicesList.getSelectedIndex());
          discoveryAgent.searchServices(attrSet, uuidSet, remoteDevice ,
 this);

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


  public void servicesDiscovered(int transID, ServiceRecord[]
        servRecord){

      for(int i = 0; i < servRecord.length; i++) {

          DataElement serviceNameElement =
                 servRecord[i].getAttributeValue(0x0100);
          String _serviceName = (String)serviceNameElement.getValue();
          String serviceName = _serviceName.trim();
          btConnectionURL =
                 servRecord[i].getConnectionURL (ServiceRecord.
          NOAUTHENTICATE_NOENCRYPT, false);

      }
  display.setCurrent(readyToConnectForm);
  readyToConnectForm.append("\n\nNote: la conexión URL es: " +
        btConnectionURL);

  }

  public void serviceSearchCompleted(int transID, int respCode) {

      if (respCode == DiscoveryListener.SERVICE_SEARCH_COMPLETED) {
          // el proceso de búsqueda de servicio fue satisfactorio

      } else {
          // el proceso de búsqueda de servicio ha fallado
      }

  }
```

```
   }
////////////////

    /**
     * Esta clase se usa para leer datos de un dispositivo RFCOMM
     */

class COMMReader extends Thread {

       public COMMReader() {

       }

       public void run(){

              try{
                     StreamConnection connection =
       (StreamConnection)Connector.open(btConnectionURL);

                     //
                     // abre una conexión de entrada para obtener algunos datos
                     //con el mecanismo de conexión SPP

       InputStream in = connection.openInputStream();

       StringBuffer out = new StringBuffer();
       byte[] b = new byte[4096];
       for (int n; (n = in.read(b)) != -1;) {
                out.append(new String(b, 0, n));
                  }
       String micadena=out.toString();

       int tam;
       tam=micadena.length();
       int nr=0; //contador de nº de registros
       char[] temp=new char[4]; //temperatura
       micadena.getChars(0,4,temp,0);
       String tempe=new String(temp);

       char[] bat=new char[6];//porcentaje de bateria, viene al inicio de la
       cadena de datos
       micadena.getChars(4,10,bat,0);
       String bateria=new String(bat);


       for (int i=0; i<tam; i++){

              int sig=i;
              int sfg=i+1;
              char[] psep=new char[sfg-sig];   //defino donde estaria la
              posible separacion de registros
              micadena.getChars(sig, sfg, psep, 0);
              String csep=new String(psep);    //caracter q podria coincidir
                     con ser el caracter de separacion


                     if (csep.equals("-")){        //si el caracter es realmente
                                                     el de separacion...

                            nr++;
                            int si=i+1;
                            int sf=i+2;
                            int di=i+3;
                            int df=i+13;
                            int hi=i+14;
```

```
                        int hf=i+19;
                        char[] nsensor=new char[sf - si];
                        micadena.getChars(si, sf, nsensor, 0);
                        String sen = new String(nsensor);
                        char[] ndia=new char[df- di];
                        micadena.getChars(di, df, ndia, 0);
                        String dia= new String(ndia);
                        char[] nhora=new char[hf- hi];
                        micadena.getChars(hi, hf, nhora, 0);
                        String hora= new String(nhora);


                        BluetoothWSNMidlet.vBD.insertRecord(sen +
                              BluetoothWSNMidlet.vBD.SEPARATOR + dia +
                  BluetoothWSNMidlet.vBD.SEPARATOR + hora +
                        BluetoothWSNMidlet.vBD.SEPARATOR);


                                    }
                                }
      dataViewForm.append( "\n" + nr + " registros nuevos cargados
      \n\nInformación de la fuente Bluetooth Puerta Pasarela: \n Temperatura:
      " + tempe + " \nNivel de bateria: " + bateria );

            in.close();
                  connection.close();

             }
             catch(IOException ioe){
                  ioe.printStackTrace();
             }
      }
    }
}
```

7.1.2 MenuPrincipal

```
package wsn;

import javax.microedition.lcdui.*;

import wsn.BluetoothWSNMidlet;

/**
  * <p>Descripción: Menu de opciones de la aplicacion</p>
  */

public class MenuPrincipal
    extends List
    implements CommandListener {
      public Display display;
  private static final String[] MENU_CHOICES = new String[3];

  static {

    MENU_CHOICES[0] = "Consultar dia";

    MENU_CHOICES[1] = "Consultar Grupo contador";

    MENU_CHOICES[2] = "Carga manual";

  }
```

```
  // --- Variables ------------------------------------------------//

  private ConsumoDisplay vConsum= new ConsumoDisplay(this);
  private ConsultaDisplay vConsul = new ConsultaDisplay(this);
  private CargaDisplay vCarg = new CargaDisplay(this);


  private Command CMD_EXIT = new Command("Salir", Command.EXIT, 1);

  /** Constructor */
  public MenuPrincipal() {
    super("Menu principal", List.IMPLICIT, MENU_CHOICES, null);
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  /**
   * Inicializar el Menu

   */

  private void jbInit() throws Exception {

    setCommandListener(this);
    //
    addCommand(CMD_EXIT);

  }

  public void commandAction(Command pCommand, Displayable pDisplayable) {

    if (pCommand == CMD_EXIT) {

    }
    else {

      String selectedItem = getString(getSelectedIndex());

      if (selectedItem.equals(MENU_CHOICES[0])) {
        //
        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vConsum);
      }
      else if (selectedItem.equals(MENU_CHOICES[1])) {
        //
        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vConsul);
      }
      else if (selectedItem.equals(MENU_CHOICES[2])) {
        //
        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vCarg);
      }


    }
  }
}
```

7.1.3 CargaDisplay

```
package wsn;

import javax.microedition.lcdui.*;

import wsn.BluetoothWSNMidlet;

import wsn.AlertaConfirmacion;
/**
 * <p>Título: CargaDisplay</p>
 * <p>Descripción: Interfaz para la creación de registros en el repositorio
via bluetooth</p>
   <p>
  */
//Se cargara en la base de datos un registro manualmente. Cada registro
contiene el nºde Grupo Contador, el dia y la hora (en horas y minutos).

public class CargaDisplay
    extends Form
    implements CommandListener {

  private Displayable vMenuPrincipal;
  TextField nSensor;
  TextField Fecha;
  TextField horaymin;


  protected Command CMD_BACK = new Command("Back", Command.BACK, 1);
  protected Command CMD_OK = new Command("OK", Command.OK, 1);
  protected Command CMD_SHOW = new Command("Mostrar", Command.SCREEN, 1);

  /**
   * Constructor
   *
   */
  public CargaDisplay(String pTitulo) {
    super(pTitulo);
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }

  }

  /**
   * Constructor
   *
   * @param pOrigDisp Display where to get back when finished
   */
  public CargaDisplay(Displayable pOrigDisp) {
    this("Carga Display");
    this.vMenuPrincipal = pOrigDisp;
    setCommandListener(this);

    //
    this.addCommand(CMD_OK);
    this.addCommand(CMD_BACK);
  }

  /**
```

```
    * Component initialization
    *
    * @throws java.lang.Exception
    */
  private void jbInit() throws Exception {
    //
        nSensor = new TextField("Nº Grupo Contador", "1", 1,
TextField.PHONENUMBER);
        nSensor.setLayout(Item.LAYOUT_CENTER | Item.LAYOUT_TOP);
        nSensor.setPreferredSize(127, 42);

        Fecha = new TextField("Dia#Mes#Año", "01#01#2008", 10,
TextField.PHONENUMBER);
        Fecha.setLayout(Item.LAYOUT_CENTER | Item.LAYOUT_TOP);
        Fecha.setPreferredSize(127, 42);


        horaymin = new TextField("Hora*Min", "00*00", 5,
TextField.PHONENUMBER);
        horaymin.setLayout(Item.LAYOUT_CENTER | Item.LAYOUT_TOP);
        horaymin.setPreferredSize(127, 42);


        this.append(nSensor);
        this.append(Fecha);
        this.append(horaymin);
   }

  /**
   *
   * Manejo de eventos de comandos
   *
   * @param pCommand Comando ejecutado
   * @param pDisplayable Display where event happened
   */

  public void commandAction(Command pCommand, Displayable pDisplayable) {
Alert vAlert;

    if (pCommand == CMD_OK) {

      if (validateFields()) {


       BluetoothWSNMidlet.vBD.insertRecord(nSensor.getString() +
       BluetoothWSNMidlet.vBD.SEPARATOR +
       Fecha.getString()+ BluetoothWSNMidlet.vBD.SEPARATOR
        + horaymin.getString() +  BluetoothWSNMidlet.vBD.SEPARATOR);


       vAlert = new AlertaConfirmacion("Se ha insertado con éxito el
       registro");

       Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
      }
    }
else if (pCommand == CMD_BACK) {

Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vMenuPrincipal);
    }
  }
```

```
  /**
   * Check that all fields has been fulfilled
   *
   * @return true if all fields have been fulfilled
   *         false in any other case
   */

 protected boolean validateFields() {
 if ( (nSensor.getString().length() == 0) ||
      (Fecha.getString().length() == 0)||(horaymin.getString().length() == 0))

     {Alert vAlert = new AlertaError(
          "Debe informar de todos los campos del registro");
      Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
      return false;
    }
    else {
      return true;
    }
  }
}
```

7.1.4 ConsumoDisplay

```
package wsn;

import javax.microedition.lcdui.*;
import wsn.AlertaError;
import wsn.ListadoResultado;
import wsn.ListadoResultadoble;
import wsn.BluetoothWSNMidlet;

/**
 * <p>Título: ConsumoDisplay</p>
 * <p>Descripción: Interfaz de consulta de consumo</p>*/

public class ConsumoDisplay
    extends Form
    implements CommandListener {

  protected Displayable vMenuPrincipal;
  protected Command CMD_BACK = new Command("Back", Command.BACK, 1);
  protected Command CMD_OK = new Command("OK", Command.OK, 1);


  TextField textField1;  //Consumo diario

  String[] vOptionStrings = {
              "Todos","1-S.Desiderio", "2-S.Lorenzo", "3-S.Giobatta"};
  ChoiceGroup choiceGroup1;

  //ImageItem vImageItem1;

  /**
   * Constructor
   *
   */

 public ConsumoDisplay(Displayable pOrigDisp) {
    super("Consumo Eléctrico");
    this.vMenuPrincipal = pOrigDisp;
    try {
      jbInit();
```

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

  /**
   * Constructor
   *
   */
  public ConsumoDisplay(Displayable pOrigDisp, String pTitulo) {
    super(pTitulo);
    this.vMenuPrincipal = pOrigDisp;
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }

  }
  /**
   * Inicializacion de componentes
   *
   * @throws java.lang.Exception
   */
  private void jbInit() throws Exception {
    textField1 = new TextField("Dia:\n     DD#MM#AAAA", "", 10,
    TextField.PHONENUMBER);
    textField1.setPreferredSize(141, 44);

    choiceGroup1 = new ChoiceGroup("Grupo Contador:",ChoiceGroup.EXCLUSIVE,
    vOptionStrings, null);

    choiceGroup1.setPreferredSize(145, 68);

    setCommandListener(this);

    addCommand(CMD_OK);
    addCommand(CMD_BACK);


    this.append(textField1);
    this.append(choiceGroup1);

  }

  /**
    * Manejor de eventos de comandos
    *
    * @param pCommand Comando ejecutado
    * @param pDisplayable Display donde el evento ocurre
    */
  public void commandAction(Command pCommand, Displayable pDisplayable) {

    int vFilter ;

    if (pCommand == CMD_OK) {

        int[] vClavesLeidos=null;

      switch (choiceGroup1.getSelectedIndex()) {

      case (1)://solo registros del grupo de contadores 1-S.Desiderio
```

```
 vClavesLeidos = BluetoothWSNMidlet.vBD.searchRecord(2,
textField1.getString());

if ( (vClavesLeidos == null) || (vClavesLeidos.length == 0)) {
  Alert vAlert = new AlertaError(
      "No existen datos para la consulta realizada ");
  Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
}
else {
  // Crear una ventana de listado pasándole los datos de filtrado

    ListadoResultadoble vListadoble = new ListadoResultadoble(this,
"Lista de registros", 2, textField1.getString(),1,"1",
BluetoothWSNMidlet.vBD);

Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vListadoble);

   }
 break;


case (2)://solo registros del grupo de contadores 2-S.Lorenzo

 vClavesLeidos = BluetoothWSNMidlet.vBD.searchRecord(2,
textField1.getString());

  if ( (vClavesLeidos == null) || (vClavesLeidos.length == 0)) {
    Alert vAlert = new AlertaError(
        "No existen datos para la consulta realizada ");
    Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
  }
  else {
    // Crear una ventana de listado pasándole los datos de filtrado

      ListadoResultadoble vListadoble = new ListadoResultadoble(this,
"Lista de registros", 2, textField1.getString(),1,"2",
BluetoothWSNMidlet.vBD);

Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vListadoble);

     }
break;

case (3)://solo registros del grupo de contadores 3-S.Giobatta

vClavesLeidos = BluetoothWSNMidlet.vBD.searchRecord(2,
textField1.getString());

  if ( (vClavesLeidos == null) || (vClavesLeidos.length == 0)) {
    Alert vAlert = new AlertaError(
        "No existen datos para la consulta realizada ");
    Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
  }
  else {
    // Crear una ventana de listado pasándole los datos de filtrado

      ListadoResultadoble vListadoble = new ListadoResultadoble(this,
"Lista de registros", 2, textField1.getString(),1,"3",
BluetoothWSNMidlet.vBD);

Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vListadoble);

     }
```

```
            break;

         default: //mostrar todos los contadores
             vFilter=2;
             vClavesLeidos = BluetoothWSNMidlet.vBD.searchRecord(2,
                    textField1.getString());

             //comprueba si hay algun registro de contadores para ese dia
             if ( (vClavesLeidos == null) || (vClavesLeidos.length == 0)) {
               Alert vAlert = new AlertaError(
                   "No existen datos para la consulta realizada ");

       Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
             }
             else {
               // Crear una ventana de listado pasándole los datos de filtrado

               ListadoResultado vListado = new ListadoResultado(this, "Lista de
       registros", vFilter, textField1.getString(), BluetoothWSNMidlet.vBD);

       Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vListado);

                }

         }


       }

     else if (pCommand == CMD_BACK) {

    Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vMenuPrincipal);
     }

  }
  }
```

7.1.5 ConsultaDisplay

```
package wsn;

import javax.microedition.lcdui.*;

import wsn.AlertaError;
import wsn.ListadoResultado;

import wsn.BluetoothWSNMidlet;

/**
 * <p>Título: ConsultaDisplay</p>
 * <p>Descripción: Interfaz de consulta de registros</p>*/

public class ConsultaDisplay
    extends Form
    implements CommandListener {

  protected Displayable vMenuPrincipal;
  protected Command CMD_BACK = new Command("Back", Command.BACK, 1);
  protected Command CMD_OK = new Command("OK", Command.OK, 1);

  TextField textField1;  //sensor
```

```
  String[] vOptionStrings = {
      "Sin filtro,mostrar todo", "Grupo Contador"};
  ChoiceGroup choiceGroup1;

  //ImageItem vImageItem1;

  /**
   * Constructor
   *
   * @param pOrigDisp Display al que hay que volver cuando finalice
   */
  public ConsultaDisplay(Displayable pOrigDisp) {
    super("Consulta detallada");
    this.vMenuPrincipal = pOrigDisp;
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  /**
   * Constructor
   *
   * @param pOrigDisp  Display al que hay que volver cuando finalice
   * @param pTitulo Titulo del Display
   */
  public ConsultaDisplay(Displayable pOrigDisp, String pTitulo) {
    super(pTitulo);
    this.vMenuPrincipal = pOrigDisp;
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }

  }
  /**
   * Inicializacion de componentes
   *
   * @throws java.lang.Exception
   */
  private void jbInit() throws Exception {
    // Set up this Displayable to listen to command events
    textField1 = new TextField("nº de Grupo Contador", "", 1,
TextField.NUMERIC);
    textField1.setPreferredSize(141, 44);

    choiceGroup1 = new ChoiceGroup("Buscar por ...",
                                   ChoiceGroup.EXCLUSIVE, vOptionStrings,
null);
    choiceGroup1.setPreferredSize(145, 68);

    setCommandListener(this);

    addCommand(CMD_OK);
    addCommand(CMD_BACK);

    this.append(choiceGroup1);
    this.append(textField1);
```

```
    }

  /**
    * Manejor de eventos de comandos
    *
    * @param pCommand Comando ejecutado
    * @param pDisplayable Display donde el evento ocurre
    */
  public void commandAction(Command pCommand, Displayable pDisplayable) {

    int vFilter = 0;

    if (pCommand == CMD_OK) {

      switch (choiceGroup1.getSelectedIndex()) {
        case (1):
          vFilter = 1;
          break;

        default:
          vFilter = -1;
      }
      // Recuperar claves y contenido de los registros

      int[] vClavesLeidos=null;
      if (vFilter==1 ||vFilter==-1 ){

             vClavesLeidos = BluetoothWSNMidlet.vBD.searchRecord(vFilter,
      textField1.getString());
       }

      if ( (vClavesLeidos == null) || (vClavesLeidos.length == 0)) {
        Alert vAlert = new AlertaError(
            "No existen datos para la consulta realizada ");
        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
      }
      else {
        // Crear una ventana de listado pasándole los datos de filtrado
            if (vFilter==1 ||vFilter==-1 ){
       vFilter=1; //para q cuando muestre todos los registros (-1) muestre los
       resultados por dias (2)
        ListadoResultado vListado = new ListadoResultado(this, "Lista de
        registros", vFilter, textField1.getString(), BluetoothWSNMidlet.vBD);
        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vListado);
             }

          }
     }
    else if (pCommand == CMD_BACK) {

Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vMenuPrincipal);
    }

  }
  }
```

7.1.6 BDatos

```
package wsn;

import javax.microedition.rms.*;

/**
  * <p>Descripción: Clase para gestionar un repositorio de registros RMS</p>*/

public class BDatos {

  private String vActiveStore = null;
  private RecordStore vRecStore;
  private BDFiltro vFilter = null;
  private RecordEnumeration vEnum = null;
  public static String SEPARATOR = ":";

  /**
   * Constructor sin parámetros que genera un almacén de datos llamado
'Predeterminado'
   */
  public BDatos() {
    createStore("Predeterminado");
    vActiveStore = "Predeterminado";
  }

  /**
       * Constructor que permite crear un almacén de datos con el nombre
   *   proporcionado como parámetro
   *
   * @param pStoreName Nombre del almacén de datos a generar
   */
  public BDatos(String pStoreName) {
    createStore(pStoreName);
    vActiveStore = pStoreName;
  }

  /**
       * Activa el repositorio de registros con el nombre proporcionado como
   *  parámetro.
   * Si el repositorio no existe genera un nuevo repositorio
   *
   * @param pStoreName Nombre del repositorio a abrir
   */
  public void createStore(String pStoreName) {
    try {
      vRecStore = RecordStore.openRecordStore(pStoreName, true);
      vActiveStore = pStoreName;
    }
    catch (Exception e) {

    }
  }

  /**
   * Elimina el repositorio de registros activo.
   */
  public void deleteStore() {
    try {
      RecordStore.deleteRecordStore(vActiveStore);
    }
    catch (Exception e) {
```

```
  }
}

/**
 * Método para recuperar el filtro actual del repositorio de registros
 *
 * @return El filtro activo sobre el repositorio de registros
 */
public BDFiltro getFilter() {
  return vFilter;
}

/**
 * Crea el filtro del repositorio
 *
 * @param pPos Posición del campo dentro del registro que se va a comparar
 * @param pValue Valor con el que se compara la posición del registro
 */
public void setFilter(int pPos, String pValue) {
  vFilter = new BDFiltro(pPos, pValue);
}

/**
 * Elimina el filtro del repositorio
 */
public void resetFilter() {
  vFilter = null;
}

/**
 * Busca en el repositorio de registros los que coinciden con el valor
 * definido en pValue para la posición dentro del registro indicada por pPos
 *
 * @param pPos Posición del campo dentro del registro (empezando en 0) que
 * se compara.
 *  Si pPos toma el valor -1, se devuelven los identificadores de todos los
 *  registros.
 *
 * @param pValue Valor dentro del registro para el que buscamos los
 * registros
 * @return Vector de enteros con los identificadores de los registros que
 * coinciden para el valor en la posición dada como parámetro
 */
public int[] searchRecord(int pPos, String pValue) {

  try {
    if (pPos != -1) {
      setFilter(pPos, pValue);
      vEnum = vRecStore.enumerateRecords(vFilter, null, false);
    }
    else {
      resetFilter();
      vEnum = vRecStore.enumerateRecords(null, null, false);
    }

    int[] vResult = new int[vEnum.numRecords()];
    int i = 0;
    while (vEnum.hasNextElement()) {
      vResult[i] = vEnum.nextRecordId();
      i++;
    }
    return vResult;
  }
  catch (RecordStoreException eStoreException) {
```

```
      return null;
    }
    finally {
      resetFilter();
    }
  }

  /**
   * Devuelve un array de registros (como cadena) correspondientes al array de
   * identificadores de registro que se proporcionan mediante parámetro
   *
   * @param pArrId Array de identificadores de los registros a recuperar
   * @return Array de registros (como string) correspondientes al array de
   * identificadores de registro dados como parametro
   */

  public String[] retrieveData(int[] pArrId) {
    String[] vResult = new String[pArrId.length];
    try {
      for (int i = 0; i < pArrId.length; i++) {
        vResult[i] = new String(vRecStore.getRecord(pArrId[i]));
      }
      return vResult;
    }
    catch (RecordStoreException eStoreException) {
      eStoreException.printStackTrace();
      return null;
    }
  }

  /**
   * Elimina un conjunto de registros del repositorio asociados al array de
   * identificadores
   * que se pasa como parámetro
   *
   * @param pArrId Array de identificadores de los registros a recuperar
   * @return Cadena con los identificadores de registro eliminados
   *
   */
  public String deleteData(int[] pArrId) {
    StringBuffer vResult = new StringBuffer();
    try {
      for (int i = 0; i < pArrId.length; i++) {
        vRecStore.deleteRecord(pArrId[i]);
        vResult.append(pArrId[i]);
      }

      return vResult.toString();
    }
    catch (RecordStoreException eStoreException) {
      eStoreException.printStackTrace();
      return null;
    }
  }

  /**
   * Elimina el registro identificada por el ID proporcionado como parámetro
   *
   * @param pArrId   Identificador del registro
   * @return pArrId  Si la operación termina correctamente <br\>
   *         -1      e.o.c.
   */
  public int deleteData(int pArrId) {
```

```
    try {
      vRecStore.deleteRecord(pArrId);

      return pArrId;
    }
    catch (RecordStoreException eStoreException) {
      eStoreException.printStackTrace();
      return -1;
    }
  }

  /**
   * Insertar un registro en el repositorio
   *
   * @param pData Cadena con los datos correspondientes al registro
   *
   */
  public void insertRecord(String pData) {
    try {
      vRecStore.addRecord(
          pData.getBytes(), // The string bytes.
          0, // Record offset.
          pData.getBytes().length); // Number of bytes to store.
    }
    catch (RecordStoreNotOpenException pNotOpened) {

      pNotOpened.printStackTrace();
    }
    catch (RecordStoreFullException pStoreFull) {

      pStoreFull.printStackTrace();
    }
    catch (RecordStoreException pStoreException) {

      pStoreException.printStackTrace();
    }
  }

  /**
   * Cierra el repositorio con el que estábamos trabajando
   */
  public void closeStore() {
    try {
      vRecStore.closeRecordStore();
    }
    catch (RecordStoreNotOpenException pNotOpened) {

      pNotOpened.printStackTrace();
    }
    catch (RecordStoreException pStoreException) {

      pStoreException.printStackTrace();
    }
  }

  /**
   * Recupera el valor de una columna del registro del repositorio.
   * Las columnas se separan mediante el caracter definido en la constante
   * de clase SEPARATOR
   *
   * @param pRow Cadena del registro a procesar
   * @param pPos Posición del registro a recuperar. Las distintas posiciones
   *         se separan mediante el caracter SEPARATOR
   *
```

```
   * @return Cadena vacía("") si la posición es incorrecta o no hay cadena
   * "SEPARATOR" <br\>
   *         Una cadena con el valor en la posición pPos dentro del registro
   */
  public String getColumn(String pRow, int pPos) {
    int vField = 1;
    int vLastIdx = 0;
    // Busca la primera ocurrencia de "separación"
    int vIdx = pRow.indexOf(SEPARATOR, 0);
    if ( (pPos < 1) || (vIdx == -1)) {
      return "";
    }
    // Itera hasta encontrar una ocurrencia de separacion o un final de cadena

    while (vField < pPos) {
      vLastIdx = vIdx + 1;
      vIdx = pRow.indexOf(SEPARATOR, vLastIdx);
      // Encuentro de final de cadena
      if (vIdx == -1) {
        return "";
      }
      vField++;
    }
    return pRow.substring(vLastIdx, vIdx);
  }
}
```

7.1.7 BDFiltro

```
package wsn;

import javax.microedition.rms.*;

/**
 *
 * <p>Título: DBFilter</p>
 * <p>Descripción: Filtro para el repositorio de registros que nos permite
 * definir qué registros se recuperarán al realizar una búsqueda</p>
 */

public class BDFiltro implements RecordFilter {
  int vPos;
  String vValue;

  /**
   * Constructor
   *
   * @param pPos Columna del registro sobre la que aplicar el filtro
   * @param pValue Valor con el que se compara la columna del registro
   */
  public BDFiltro(int pPos, String pValue) {
    vPos = pPos;
    vValue = pValue;
  }

  /**
   * Método que define la condición que determina si se devuelve un registro
   *
   * @param pData registro a procesar
   * @return true si la columna definida en el atributo vPos tiene el valor
   * definido en vValue .false si no coinciden
   */
```

```
public boolean matches(byte[] pData) {
    String vData = new String(pData);

    int vField = 1;
    int vLastIdx = 0;
    // Busca la primera ocurrencia de separacion
    int vIdx = vData.indexOf(BDatos.SEPARATOR, 0);

    if (vIdx == -1) {
      return false;
    }
    // Itera con una ocurrencia apropriada hasta un fin apropiado
    //o una conclusion de cadena
    while (vField < vPos) {
      vLastIdx = vIdx + 1;
      vIdx = vData.indexOf(BDatos.SEPARATOR, vLastIdx);
      // Fin de cadena encontrado
      if (vIdx == -1) {
        return false;
      }
      vField++;
    }
    if (vValue.equals(vData.substring(vLastIdx, vIdx))) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

7.1.8 ListadoResultado

```
package wsn;

import wsn.AlertaConfirmacion;
import javax.microedition.lcdui.*;
import wsn.VerDetalleDisplay;
import wsn.BluetoothWSNMidlet;

/**

 * <p>Descripción: Muestra una lista de entradas y permite ejecutar una serie
 * de operaciones sobre una o varias entradas</p>

 */

public class ListadoResultado
    extends Form
    implements CommandListener {

  Displayable vParentDisplay; //Ventana a la que volveremos desde esta ventana
  int vFilter;                //Campo por el que se filtra
  String vFilterString;   //Cadena para filtrar los resultados que se muestran
                          //en el listado
  BDatos vDataBase;        //Base de datos para refrescar el contenido.

  // Operaciones a ejecutar sobre el listado
  Command CMD_BACK = new Command("Atrás", Command.BACK, 1);
  Command CMD_OP1 = new Command("Borrar", Command.SCREEN, 1);
  Command CMD_OP2 = new Command ("Consumo", Command.SCREEN,1);
  Command CMD_OP3 = new Command("Deseleccionar todos", Command.SCREEN, 1);
  Command CMD_OP4 = new Command("Seleccionar todos", Command.SCREEN, 1);
```

```
Command CMD_OP5 = new Command("Ver Detalle", Command.SCREEN, 1);

// Ventanas de acceso a la ventana


ChoiceGroup choiceGroup1 = null;
StringItem vStringItem1 = null;
String[] vResultStrings = null;
int[] vResultKeys = null;
private VerDetalleDisplay vConsul;


/**
 * Constructor
 *
 * @param pOrigDisp Display al que volvemos con el comando de vuelta atrás
 * @param pTitulo   Título de la ventana
 * @param pFilter   Filtro para la búsqueda
 * @param pFilterString Cadena a utilizar en el filtro
 * @param pDataBase Base de datos de la que se leen los datos
 */
public ListadoResultado(Displayable pOrigDisp,
                        String pTitulo,
                        int pFilter,
                        String pFilterString,
                        BDatos pDataBase) {
   super(pTitulo);
   this.vParentDisplay = pOrigDisp;
   try {
     vFilter = pFilter;
     vFilterString = pFilterString;
     vDataBase = pDataBase;
     vResultKeys = vDataBase.searchRecord(vFilter, vFilterString);
     vResultStrings = pDataBase.retrieveData(vResultKeys);

     //Filtrar las cadenas recuperadas
     for (int i=0; i<vResultStrings.length; i++) {
         vResultStrings[i]=vDataBase.getColumn(vResultStrings[i],3-
vFilter);//si filtro campo 1, muestro la etiqueta del 2 y viceversa
       }
     jbInit();
   }
   catch (Exception e) {
     e.printStackTrace();
   }

 }


/**
 * Component initialization
 *
 * @throws java.lang.Exception
 */
private void jbInit() throws Exception {

  choiceGroup1 = new ChoiceGroup("Resultados",
                                 ChoiceGroup.MULTIPLE,
                                 vResultStrings,
                                 null);

choiceGroup1.setPreferredSize(145, 68);
```

```
    setCommandListener(this);

    addCommand(CMD_OP5);
    addCommand(CMD_OP4);
    addCommand(CMD_OP2);
    addCommand(CMD_OP3);
    addCommand(CMD_OP1);
    addCommand(CMD_BACK);
    this.append(choiceGroup1);
  }

  /**
   *
   * Manejo de eventos de comandos
   *
   * @param pCommand Comando ejecutado
   * @param pDisplayable Muestra donde ocurrio el evento
   */
  public void commandAction(Command pCommand, Displayable pDisplayable) {
    try {
       Alert vAlert;
      // Borrar comando
      if (pCommand == CMD_OP1) {
        // Pantalla de dialogo que pregunta confirmacion de operacion
        DialogScreen dl = new DialogScreen(
            "Confirmacion",
            "¿Desea eliminar los datos seleccionados?",
            Image.createImage(Image.createImage("/query.png")),
            DialogScreen.CONFIRMAR | DialogScreen.CANCELAR,
            BluetoothWSNMidlet.instance,

            new NewDialogListener() {
              public void onCONFIRMAR() {
                int vTotal = choiceGroup1.size();
                int vDecTotal = vTotal;

                // Borrar datos seleccionados
                for (int i = 0; i < vTotal; i++) {
                  if (choiceGroup1.isSelected(i)) {
                    // Si el elemento estaba marcado, se elimina.
                    //Procesar según CMD_OP1 el elemento


                      BluetoothWSNMidlet.vBD.deleteData(vResultKeys[i]);
                    vDecTotal--;
                  }
                }
                // Si toda la lista de elementos ha sido borrada, notificar
                // que la lista esta vacia.
                if (vDecTotal==0) {
                  delete(0);
                  vStringItem1 = new StringItem("Resultados\n\nNo quedan
                        elementos disponibles", "");
                  append(vStringItem1);
                  removeCommand(CMD_OP1);


                } else {
                  // Actualiza la lista con los datos restantes
                  refreshValues();
                }
              }
            }
```

```
        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(dl);

        // Modificacion de proceso y vista detallada.
      } else if (pCommand == CMD_OP5) {
        int count = 0;
        for (int i = 0; i < choiceGroup1.size(); i++) {
          if (choiceGroup1.isSelected(i)) {
             count++;
          }
        }
        // Solo se puede detallar un elemento
        // Notificar si se seleccionan más
        if (count > 1) {
          Alert vAlert1 = new AlertaError("Hay más de un elemento
          seleccionado.\nSeleccione solo uno.");
          Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert1);

         } else if (count == 0) {
          Alert vAlert1 = new AlertaError("Debe seleccionar algún elemento.");
          Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert1);

         } else {
          int i = 0;
          while ((i < choiceGroup1.size()) && (!choiceGroup1.isSelected(i))){
                i++;
          }
          // devuelve informacion del elemento
          if (choiceGroup1.isSelected(i)) {
            int[] vResult = new int[1];
            vResult[0] = vResultKeys[i];

            if (pCommand == CMD_OP5) {
             this.vConsul = new VerDetalleDisplay(this, vResult);

          Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vConsul);
            }
          }
        }
                // Deseleccionar todos los elementos de la lista
        }

      else if (pCommand == CMD_OP3) {
        for (int i = 0; i < choiceGroup1.size(); i++) {
          choiceGroup1.setSelectedIndex(i, false);
        }
       //      Select all elements in the list
      } else if (pCommand == CMD_OP4) {
        for (int i = 0; i < choiceGroup1.size(); i++) {
          choiceGroup1.setSelectedIndex(i, true);
        }

      }else if (pCommand == CMD_OP2) {
         int num= choiceGroup1.size();


          vAlert = new AlertaConfirmacion("Consumo eléctrico:\n" + (num * 0.25)
+ " KW/h");
          Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
          // Return to the vParentDisplay
       }

        else if (pCommand == CMD_BACK) {

Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vParentDisplay);
```

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

  /**
   * Actualiza todos los valores de la lista después de las modificaciones
   */
  public void refreshValues(){
    vResultKeys = vDataBase.searchRecord(vFilter, vFilterString);
    vResultStrings = vDataBase.retrieveData(vResultKeys);
    for (int i = 0; i < vResultStrings.length; i++) {
      vResultStrings[i] = vDataBase.getColumn(vResultStrings[i], (3-vFilter));
    }

    // Si no quedan elementos en la lista...notificarlo
    delete(0);
    if (vResultKeys.length==0 ) {
      vStringItem1 = new StringItem("Resultados\n\n No quedan elementos
                                    disponibles", "");
      append(vStringItem1);
      removeCommand(CMD_OP1);

    // Carga los elementos en la lista
    } else {
      choiceGroup1 = new ChoiceGroup("Resultados",
                                    ChoiceGroup.MULTIPLE,
                                    vResultStrings,
                                    null);
      choiceGroup1.setPreferredSize(145, 68);
      append(choiceGroup1);
    }
  }
}
```

7.1.9 ListadoResultadoble

```
package wsn;

import wsn.AlertaConfirmacion;
import javax.microedition.lcdui.*;
import wsn.VerDetalleDisplay;
import wsn.BluetoothWSNMidlet;

/**

 * <p>Descripción: Muestra una lista de entradas y permite ejecutar una serie
 * de operaciones sobre una o varias entradas</p>

 */

public class ListadoResultadoble
    extends Form
    implements CommandListener {

  Displayable vParentDisplay; //Ventana a la que volveremos desde esta ventana
  int vFilter;
  String vFilterString;
  int vFilter2;//Campo por el que se filtra

  String vFilterString2;//Cadena para filtrar los resultados que se muestran
                        //en el listado
```

```
BDatos vDataBase;        //Base de datos para refrescar el contenido.

 // Operaciones a ejecutar sobre el listado
 Command CMD_BACK = new Command("Atrás", Command.BACK, 1);
 Command CMD_OP1 = new Command("Borrar", Command.SCREEN, 1);
 Command CMD_OP2 = new Command ("Consumo", Command.SCREEN,1);
 Command CMD_OP3 = new Command("Deseleccionar todos", Command.SCREEN, 1);
 Command CMD_OP4 = new Command("Seleccionar todos", Command.SCREEN, 1);
 Command CMD_OP5 = new Command("Ver Detalle", Command.SCREEN, 1);

 // Ventanas de acceso a la ventana

 ChoiceGroup choiceGroup1 = null;
 StringItem vStringItem1 = null;
 StringItem vStringItem2=null;
 String[] vResultStrings = null;
 String[] vResultStrings2 = null;
 String[] vResultStringfinal=null; // este es el grupo filtrado final que se

 mostrará en pantalla acorde al dia y grupo contador especificado
 int[] vResultKeys = null;
 int[] vResultKeys2 = null;
 int[] vResultKeysfinal = null;
 int[] vResultKeysfinalc;

 private VerDetalleDisplay vConsum;

 /**
  * Constructor
  *
  * @param pOrigDisp Display al que volvemos con el comando de vuelta atrás
  * @param pTitulo   Título de la ventana
  * @param pFilter   Filtro para la búsqueda 1
  * @param pFilterString Cadena a utilizar en el filtro
  * @param pFilter2   Filtro para la búsqueda 2
  * @param pFilterString2 Cadena a utilizar en el filtro 2
  * @param pDataBase Base de datos de la que se leen los datos
  */
 public ListadoResultadoble(Displayable pOrigDisp,
                            String pTitulo,
                            int pFilter,String pFilterString,
                            int pFilter2,String pFilterString2,
                            BDatos pDataBase) {
    super(pTitulo);
    this.vParentDisplay = pOrigDisp;
    try {
      vFilter = pFilter;
      vFilter2= pFilter2;
      vFilterString = pFilterString;
      vFilterString2= pFilterString2;
      vDataBase = pDataBase;
      vResultKeys = vDataBase.searchRecord(vFilter, vFilterString);
      vResultKeys2 = vDataBase.searchRecord(vFilter2, vFilterString2);

     int f=0;
     for (int c=0;c<(vResultKeys.length);c++){

         for (int d=0;d<(vResultKeys2.length );d++){

               if (vResultKeys[c]==vResultKeys2[d]){

                     f++;
               }
         }
```

```
      }
      vResultKeysfinalc = new int[f];
      f=0;
      for (int c=0;c<(vResultKeys.length);c++){

          for (int d=0;d<(vResultKeys2.length);d++){

                if (vResultKeys[c]==vResultKeys2[d]){
                      vResultKeysfinalc[f]=vResultKeys[c];
                      f++;
                }
          }
      }

      vResultKeysfinal=vResultKeysfinalc;

      vResultStringfinal = pDataBase.retrieveData(vResultKeysfinal);

       //Filtrar las cadenas recuperadas

       for (int i=0; i<vResultStringfinal.length; i++) {

       vResultStringfinal[i]=vDataBase.getColumn(vResultStringfinal[i],3);
//muestra los tiempos en los que se produjeron los registros de 1/4 de Kw/h
         }

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

   }

  /**
   * Component initialization
   *
   * @throws java.lang.Exception
   */
  private void jbInit() throws Exception {

    choiceGroup1 = new ChoiceGroup("Resultados",
                                   ChoiceGroup.MULTIPLE,
                                   vResultStringfinal,
                                   null);
    choiceGroup1.setPreferredSize(145, 68);

    setCommandListener(this);
    // add the commands
    addCommand(CMD_OP5);
    addCommand(CMD_OP4);
    addCommand(CMD_OP2);
    addCommand(CMD_OP3);
    addCommand(CMD_OP1);
    addCommand(CMD_BACK);
    this.append(choiceGroup1);
  }

  /**
   *
   * Manejo de eventos de comandos
   *
   * @param pCommand Comando ejecutado
   * @param pDisplayable Muestra donde ocurrio el evento
```

```
   */
  public void commandAction(Command pCommand, Displayable pDisplayable) {
    try {
       Alert vAlert;
      // Borrar comando
      if (pCommand == CMD_OP1) {
        // Pantalla de dialogo que pregunta confirmacion de operacion
        DialogScreen dl = new DialogScreen(
            "Confirmacion",
            "¿Desea eliminar los datos seleccionados?",
            Image.createImage(Image.createImage("/query.png")),
            DialogScreen.CONFIRMAR | DialogScreen.CANCELAR,
            BluetoothWSNMidlet.instance,

            new NewDialogListener() {
              public void onCONFIRMAR() {
                int vTotal = choiceGroup1.size();
                int vDecTotal = vTotal;

                // Borrar datos seleccionados
                for (int i = 0; i < vTotal; i++) {
                  if (choiceGroup1.isSelected(i)) {
                    // Si el elemento estaba marcado, se elimina.
                    //Procesar según CMD_OP1 el elemento

                      BluetoothWSNMidlet.vBD.deleteData(vResultKeysfinal[i]);
                    vDecTotal--;
                  }
                }
                // Si toda la lista de elementos ha sido borrada, notificar
                // que la lista esta vacia.
                if (vDecTotal==0) {
                  delete(0);
                  vStringItem1 = new StringItem("Resultados\n\nNo quedan
                                                      elementos disponibles",
"");
                  append(vStringItem1);
                  removeCommand(CMD_OP1);

                } else {
                  // Actualiza la lista con los datos restantes
                  refreshValues();
                }
              }
            },
            this);
        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(dl);

        // Modificacion de proceso y vista detallada.
      } else if (pCommand == CMD_OP5) {
        int count = 0;
        for (int i = 0; i < choiceGroup1.size(); i++) {
          if (choiceGroup1.isSelected(i)) {
             count++;
          }
        }
        // Solo se puede detallar un elemento
        // Notificar si se seleccionan más
        if (count > 1) {
          Alert vAlert1 = new AlertaError("Hay más de un elemento
                                              seleccionado.\nSeleccione solo
uno.");
          Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert1);
        } else if (count == 0) {
```

```
          Alert vAlert1 = new AlertaError("Debe seleccionar algún elemento.");
          Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert1);
        } else {
          int i = 0;
          while ((i < choiceGroup1.size()) && (!choiceGroup1.isSelected(i))){
                i++;
          }
          // devuelve informacion del elemento
          if (choiceGroup1.isSelected(i)) {
            int[] vResult = new int[1];
            vResult[0] = vResultKeysfinal[i];

            if (pCommand == CMD_OP5) {
             this.vConsum = new VerDetalleDisplay(this, vResult);

        Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vConsum);
            }
          }
        }

        // Deseleccionar todos los elementos de la lista
        }

      else if (pCommand == CMD_OP3) {
        for (int i = 0; i < choiceGroup1.size(); i++) {
          choiceGroup1.setSelectedIndex(i, false);
        }
       //      Select all elements in the list
      } else if (pCommand == CMD_OP4) {
        for (int i = 0; i < choiceGroup1.size(); i++) {
          choiceGroup1.setSelectedIndex(i, true);
        }

      }else if (pCommand == CMD_OP2) {
         int val= choiceGroup1.size();
          int num=0;
         for (int i = 0; i < val; i++) {
              if (choiceGroup1.isSelected(i)) {
               num++;
              }

          vAlert = new AlertaConfirmacion("Consumo eléctrico:\n" + (num *
                                          0.25) + " KW/h");
          Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vAlert);
          // Return to the vParentDisplay
       }
      }
        else if (pCommand == CMD_BACK) {

Display.getDisplay(BluetoothWSNMidlet.instance).setCurrent(vParentDisplay);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  /**
   * Actualiza todos los valores de la lista después de las modificaciones
   */
  public void refreshValues(){
}}
```

7.1.10 VerDetalleDisplay

```
package wsn;

import javax.microedition.lcdui.*;
import wsn.CambioDisplay;

/**
  * <p>Descripción: Interfaz de detalle de un registro </p>
 */

public class VerDetalleDisplay
    extends CambioDisplay {

  public VerDetalleDisplay(Displayable pAntDisp, int[] pIdTarea) {
    super("Detalle de registro", pAntDisp, pIdTarea);
    this.removeCommand(CMD_OK);
    lockFields();
  }

  public void lockFields() {
    this.nSensor.setConstraints(TextField.UNEDITABLE);

    this.horaymin.setConstraints(TextField.UNEDITABLE);
  }
}
```

7.1.11 AlertaConfirmacion

```
package wsn;

import javax.microedition.lcdui.*;

/**
 * <p>Título: AlertaConfirmacion</p>
 * <p>Descripción: Muestra un mensaje con un icono de alerta al usuario</p>
  */

public class AlertaConfirmacion
    extends Alert{

  /**
   * Constructor
   *
   * @param pText Texto para mostrar en el mensaje
   */
  public AlertaConfirmacion(String pText) {
    super(pText);
    try {
      this.setTitle("Información");
      this.setString(pText);
      this.setImage(Image.createImage("/okay.png"));
      this.setType(AlertType.CONFIRMATION);
      this.setTimeout(Alert.FOREVER);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

7.1.12 AlertaError

```
package wsn;

import javax.microedition.lcdui.*;

/**
 * <p>Título: AlertaError</p>
 * <p>Descripción: Muestra un mensaje con un icono de error al usuario</p>
  */

public class AlertaError
    extends Alert {

  Image anImage;

  /**
   * Constructor
   *
   * @param pText Texto de error para mostrar al usuario
   */
  public AlertaError(String pText) {
    super(pText);
    try {
      anImage = Image.createImage("/error.png");
      this.setTitle("Error");
      this.setString(pText);
      this.setImage(anImage);
      this.setType(AlertType.ERROR);
      this.setTimeout(Alert.FOREVER);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

7.1.13 DialogScreen

```
package wsn;

import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

/**
  * <p>Descripción: Gestor de comportamiento asociado a los botones mostrados
en el cuadro de diálogo</p>
  */
class NewDialogListener {
  public void onOK() {}
  public void onSI() {}
  public void onNO() {}
  public void onCANCELAR() {}
  public void onCONFIRMAR() {}
}

/**
  * <p>Descripción: Cuadro de dialogo asociado a un interfaz de usuario</p>
*/
```

```
public class DialogScreen
    extends Form
    implements CommandListener {

  Display vDisplayManager;
  NewDialogListener vDialogListener;
  Displayable vParentDisplay;

  public static final int OK = 1;
  public static final int SI = 2;
  public static final int NO = 4;
  public static final int CANCELAR = 8;
  public static final int CONFIRMAR = 16;

  Command cmOK;
  Command cmSI;
  Command cmNO;
  Command cmCANCELAR;
  Command cmCONFIRMAR;
  StringItem vText;
  ImageItem vImage;

  public Displayable getParentDisplay() {
    return vParentDisplay;
  }

  /**
   * Consturctor
   * @param title Título del cuadro de dialogo
   * @param text Texto a mostrar
   * @param pImage imágen a mostrar
   * @param mode Botones a mostrar
   * @param midlet Midlet asociado
   * @param pDialogListener Gestor de eventos
   * @param pParentDisplay Ventana que nos ha llevado a esta
   */
   public DialogScreen(String title,
                       String text,
                       Image pImage,
                       int mode,
                       MIDlet midlet,
                       NewDialogListener pDialogListener,
                       Displayable pParentDisplay) {
    super(title);
    vDialogListener = pDialogListener;
    vText = new StringItem(null, text);
    vText.setLayout(Item.LAYOUT_CENTER);
    vImage = new ImageItem("", pImage, Item.LAYOUT_CENTER +
    Item.LAYOUT_NEWLINE_AFTER, "Alt");
    vDisplayManager = Display.getDisplay(midlet);
    vParentDisplay = pParentDisplay;

    setCommandListener(this);

    if ( (mode & OK) != 0) {
      cmOK = new Command("OK", Command.SCREEN, 1);
      addCommand(cmOK);
    }
    if ( (mode & SI) != 0) {
      cmSI = new Command("Si", Command.SCREEN, 1);
      addCommand(cmSI);
    }
    if ( (mode & NO) != 0) {
```

```
      cmNO = new Command("No", Command.SCREEN, 1);
      addCommand(cmNO);
    }
    if ( (mode & CANCELAR) != 0) {
      cmCANCELAR = new Command("Cancelar", Command.SCREEN, 1);
      addCommand(cmCANCELAR);
    }
    if ( (mode & CONFIRMAR) != 0) {
      cmCONFIRMAR = new Command("Confirmar", Command.SCREEN, 1);
      addCommand(cmCONFIRMAR);
    }

    append(vImage);
    append(vText);

  }

  public void commandAction(Command c, Displayable s) {
    if (vDialogListener != null) {
      if (c == cmOK)
        vDialogListener.onOK();
      else if (c == cmSI)
        vDialogListener.onSI();
      else if (c == cmNO)
        vDialogListener.onNO();
      else if (c == cmCANCELAR)
        vDialogListener.onCANCELAR();
      else if (c == cmCONFIRMAR)
        vDialogListener.onCONFIRMAR();

    } else {

    }
    vDisplayManager.setCurrent(vParentDisplay);
  }
}
```

8 BIBLIOGRAFÍA Y REFERENCIAS

[1] **Wireless Sensor Networks. John A. Stankovic.** Department of Computer Science University of Virginia.Junio 2006

[2] **Redes de sensores inalámbricas. Nuevas soluciones de interconexión para la automatización industrial.** Niels Aakvaag, Jan-Erik Frey

[3] **ZigBee Alliance**, http://www.zigbee.org

[4] http://spanish.bluetooth.com/bluetooth/

[5] **Programming Wireless Devices with the Java 2 Platform, Micro Edition, Second Edition** por Roger Riggs, Antero Taivalsaari, Jim Van Peursem, Jyri Huopaniemi. Editor: Addison Wesley. Junio, 2003. ISBN: 0-321-19798-4

[6] http://java.sun.com/javame/index.jsp

[7] **J2ME: The Complete Referente.** McGraw-Hill. 2003. ISBN 0-07-222710-9

[8] **JavaTM APIs for BluetoothTM Wireless Technology (JSR82).**Motorola.Wireless Software, Applications & Services. Abril, 2002

[9] **Carbide.j 1.5 User's Guide**. Nokia.Junio, 2006

[10] **Nokia 6131 NFC SDK: Programmer's Guide .**Nokia.Version 1.1.Julio, 2007

[11] **Developing Mobile Applications with the Eclipse IDE**. Christian Kurzke. Motorola Junio 2009.

[12]http://www.forum.nokia.com/Resources_and_Information/Explore/Software_Platforms/Series_40/

[13] http://europe.nokia.com/A4307094 (Portal del Nokia 6131 NFC)

[14] http://wiki.forum.nokia.com/index.php/Portal:Java_Code_Examples