Estudio de Consumo en Redes de Sensores Inalámbricos para la detección de ondas características en ECG

Laura Gutiérrez Muñoz

Profesores Proyecto: David Atienza Alonso y
Marcos Sánchez-Elez Martín

Colaboradores:
Francisco Javier Rincón Vallejos

Curso: 2007 - 2008
Master Investigación en Informática
Facultad de Informática (UCM)

Resumen:

Las redes de sensores inalámbricas de área corporal están emergiendo como una gran solución para el seguimiento de personas con problemas de salud. Hasta ahora, estas redes se limitan a leer las señales vitales del paciente y enviar toda la información recogida a un dispositivo colector donde será posteriormente procesada o mostrada a los médicos. Este enfoque deriva en un corto tiempo de vida de los nodos que forman la red, debido al gran consumo de energía que conlleva la transmisión de todos los datos leídos a la estación base, pues la radio es el elemento que más consume del nodo. Mediante la inclusión de un algoritmo en los nodos, que procese las señales leídas por los sensores en lugar de enviarlas directamente, se consigue reducir notablemente el consumo, pues se reduce la comunicación inalámbrica, al transmitirse a la estación base sólo información relevante sobre el estado del paciente.

Para este trabajo, nos centramos en el procesamiento del electrocardiograma (ECG), usando una plataforma inalámbrica diseñada por IMEC capaz de leer 25 señales de ECG y electroencefalograma (EEG). Se ha diseñado una aplicación para el análisis de la señal ECG y el diagnóstico automático en tiempo real de patologías cardiacas, que ha sido optimizada para la escasa capacidad de procesamiento de la plataforma usada. Mediante el uso de esta aplicación se han obtenido resultados que llegan hasta el 99,11% de reducción en el consumo de energía de la radio, con respecto a otras redes en las que toda la información recogida por los sensores es transmitida a la estación base sin ningún procesamiento previo.

Abstract:

The Wireless Body Sensor Networks are emerging as a great solution for tracking people with health problems. Until now, these networks just read the patient's vital signs and send all collected information to a collector device where data will be processed or shown to doctors. This approach results in a short lifetime of nodes that make up the network, due to high energy consumption associated with the transmission of all read data to the base station, because radio is the element with higher consumption in the node. Including an algorithm to process signals read by the sensor in the nodes instead sending them directly, significantly reduce consumption, thus it reduces the wireless communication, when only relevant information about the patient state is transmitted to the base station.

For this work, we focus on electrocardiogram processing (ECG), using a wireless platform designed by IMEC that can read 25 ECG signals and electroencephalogram (EEG). An application for the analysis of the ECG signal and automatic real-time diagnosis of cardiac diseases has been designed, and it has been optimized for the limited processing capability of the platform used. Using this application, a 99.11% reduction in energy consumption of radio has been achieved, with respect to other networks, where all information collected by the sensors is transmitted to the base station without any pre-processing.

Palabras clave: Redes de Sensores Inalámbricas, Consumo, ECG, análisis de señales biomédicas

Índice:

1. Introducción **7**

- 1.1 Wireless Body Sensor Networks 7
- 1.2 Motivación 9
- 1.3 Objetivos del Estudio 10

2. Algoritmo Automático de diagnostico de ECG **13**

- 2.1 Conceptos básicos sobre ECG y ondas características 13
- 2.2 Descripción del Algoritmo de diagnostico y detección de ECG 14
- 2.3 Evolución a un Algoritmo dinámico y optimizado 17
- 2.4 Algoritmo de diagnóstico 22

3. Arquitectura de los nodos **27**

- 3.1 Arquitectura Hardware 27
- 3.2 Arquitectura Software 34
  - 3.2.1 Sistema Operativo TinyOS 36
  - 3.2.2 TOSSIM 40
  - 3.2.3 PowerTOSSIM 40

4. Protocolos de control de acceso al medio **41**

- 4.1 Introducción 41
- 4.1 Protocolo MAC en el nodo: TDMA 41

5. Estudio del Consumo **47**

- 5.1 Modificaciones para el ahorro de Consumo 48
- 5.2 Medidas y Pruebas Realizadas 50

6. Conclusiones **55**

Apéndice I: Herramientas Usadas **57**

1. Cygwin
2. Terminal
3. PowerTossim
4. Embedded Workbench IDE

Apéndice II: Manuales de instalación y uso **61**

1. Instalación de TinyOS y MSP430-GCC en Windows
2. Manual de PowerTossim
3. Manual de IAR Embedded Workbench IDE
4. Manual de NesC

7. Bibliografía **85**

1. Introducción

1.1 Wireless Body Sensor Networks (WBSN)

Las redes inalámbricas de sensores de área corporal (Wireless Body Sensor Networks [5]) están formadas por una serie de dispositivos ligeros y muy pequeños. Cada plataforma tiene uno o más sensores fisiológicos (sensores que leen las señales vitales) conectados a través de una red inalámbrica.

Esta red puede estar conectada a un servidor, base de datos, etc. que proporcione información, para realizar un análisis por los sensores, o almacene la información recopilada por éstos (incluso directamente en Internet, como se muestra en la *Fig.1.1*).

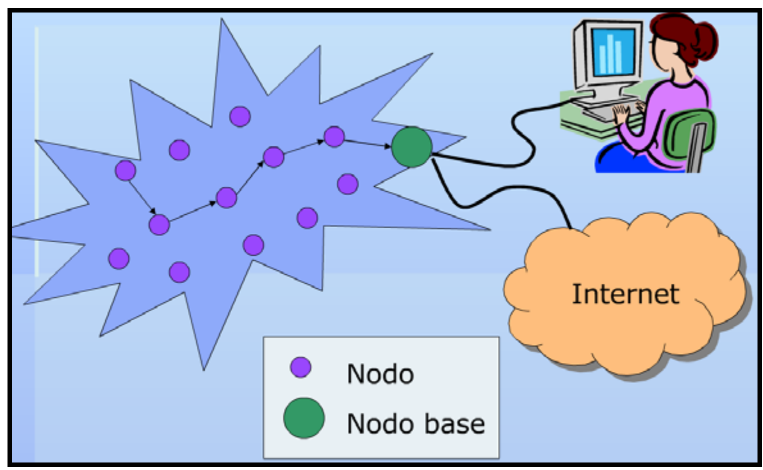


Figura. 1.1 Sistema WBSN. Formado por Sensores fisiológicos en una red Inalámbrica corporal (WBSN) que envían información a una estación base o nodo base conectado a una red local o más amplia (WAN con acceso a un servidor, base de datos, Internet, etc.).

Estos sistemas se pueden usar en multitud de campos de investigación (donde se requiera recolección de datos o muestras desde varios puntos, o diferentes mediciones),

Existen varios tipos de redes WSBN, dependiendo de la función a la que se destinan, y el tipo de sensores que incorporen, a continuación veremos una clasificación de las redes de sensores según su modo de uso:

<u>Monitorización continua</u>: los nodos que miden los mismos parámetros en un área de interés, con envío periódico de la información recogida. (Como redes WSBN destinadas a la medicina, para el seguimiento de pacientes con enfermedades crónicas)

Monitorización basada en eventos: nodos monitorizando entornos continuamente, pero solo envían información cuando ocurre algún evento (Nodos que detectan enfermedades cardiacas).

Redes Hibridas: escenarios de acción que tienen nodos de las 2 categorías anteriores.

En los nodos de la red WSBN, se diferencias varias partes, cada una con su función correspondiente (ver *Fig.1.2*):

- Constan de un sensor o varios, encargados de recoger los datos.
- Un procesador sencillo con una pequeña memoria, que permite realizar cómputos locales sobre los datos adquiridos por el sensor.
- La radio: se encarga de la comunicación inalámbrica de información a otros nodos o a la estación base.
- La fuente de alimentación del nodo, normalmente son baterías o energy scavengers o recolectores de energía del medio.

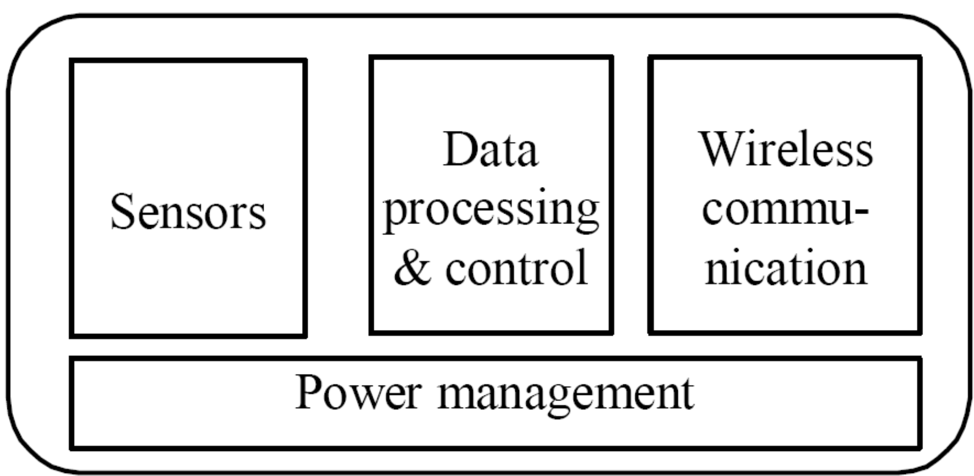


Figura 1. 2: Estructura general de las partes de un nodo.

Los nodos tienen grande limitaciones por ser de tamaño reducido, ya que deben poder colocarse en el cuerpo o ser fáciles de transportar (como diminutos parches inteligentes, integrados en la ropa, o ser implantados bajo la piel o los músculos). Así como, permitir monitorizaciones continuas y de larga duración (grandes restricciones de consumo) sin tener la necesidad de recargar sus baterías.

Las grandes limitaciones de estos nodos proponen varios retos como:
- Necesidad de bajo consumo energético, poco peso y tamaño.
- Flexibilidad de los sensores para adaptarse a las condiciones del usuario y los cambios en el entorno: por ejemplo, nodos adaptados para hacer deporte.
- Conectividad sin fallos, necesaria para la integración de los nodos dentro del sistema de monitorización.
- Comunicación y almacenamiento de datos que recoja o analice de forma segura y fiable.
- Funcionamiento de un sistema tolerante a fallos, capaz de adaptarse a los fallos de los sensores y retransmitir paquetes perdidos.

Esto dificulta la integración de aplicaciones muy complejas, que puedan funcionar en ellos, ajustándose a dichas limitaciones.

1.2 Motivación

El gran avance tecnológico de sistemas portátiles, en los últimos años, ha permitido el uso de éstos en múltiples aplicaciones, como la medicina y sus determinadas variantes.

Gracias a estos sistemas, es posible una continua monitorización biomédica, pudiendo incluso ser personalizada y automática. Asegurando la prevención de enfermedades o problemas de salud con un diagnostico temprano de cualquier anomalía en el paciente. Proporcionando así autonomía y seguridad a la persona que lleve estos dispositivos.

Las redes de sensores corporales inalámbricas (Wireless Body Sensor Networks – WBSN [5]) son una clase de estos sistemas. Formadas por dispositivos denominados nodos que permiten detectar, guardar y analizar un gran número de parámetros fisiológicos, como encefalograma (EEG), electrocardiograma (ECG), etc., que se determinarán en función del estudio al que esté dirigido el análisis medico o las características del individuo al que se le realiza.

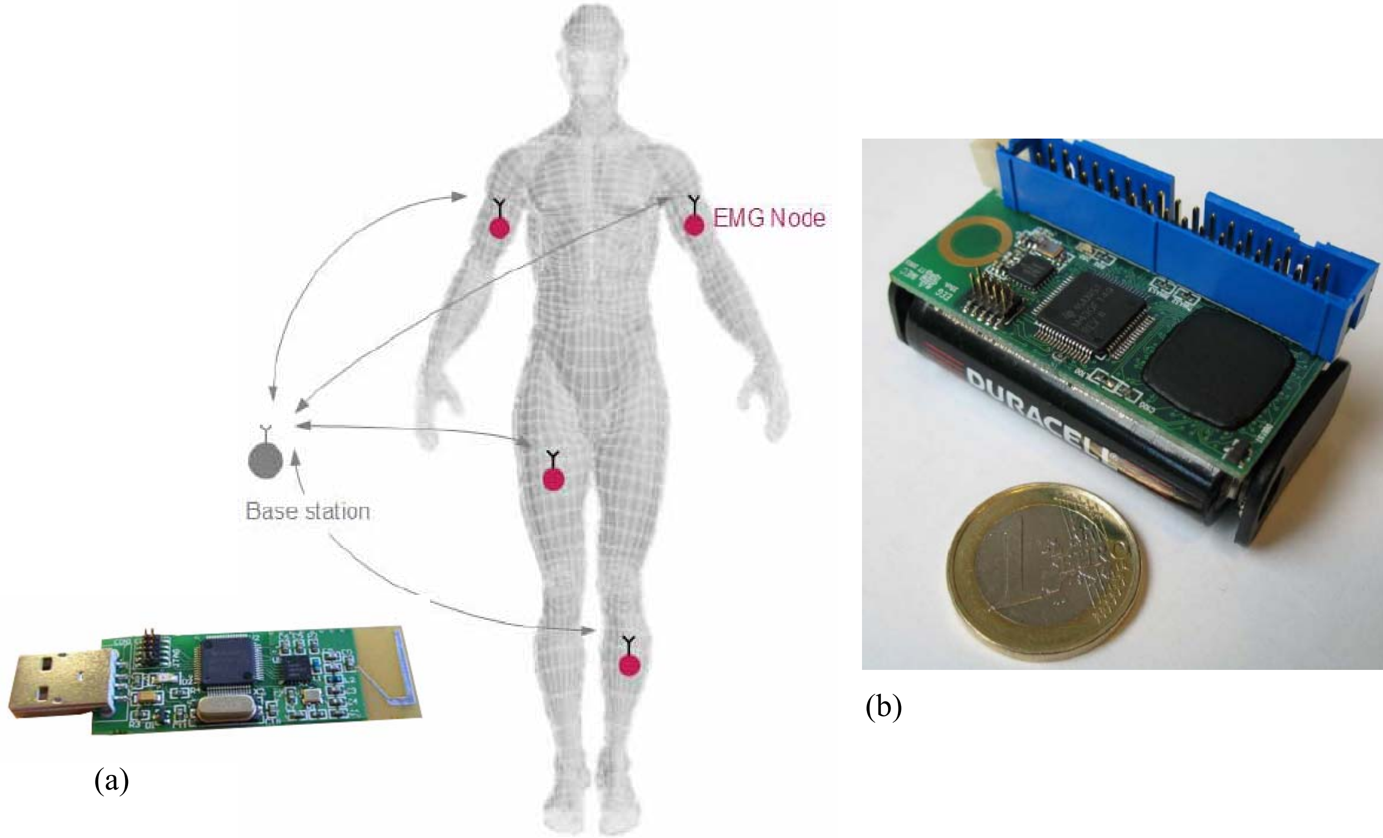


Figura 1.3: Red de Sensores Inalámbrica Corporal (WBSN).
Fotografías de una estación base (a) y un nodo (b)

Las medidas que toman los sensores se pueden enviar a una estación base (ésta puede ser un PC, PDA, teléfono móvil, etc.) para un seguimiento a distancia.

Estos nodos se deben caracterizar por su reducido tamaño (se requiere que tengan un tamaño lo suficientemente pequeño, para permitir su colocación en el cuerpo y posibilitar su uso mientras se realizan actividades físicas u otras acciones cotidianas) así como tener una gran autonomía (debido al uso de baterías, es importante la reducción de consumo de energía en ellos).

Se espera que para el 2010, la tecnología permita mejorar el modo de vida y la salud gracias a estos dispositivos, que será posible llevar en el cuerpo formando así una red corporal (WBSN). Destinando esta red de sensores a múltiples usos como el deporte, seguimiento medico, etc. Pueden proporcionar a los pacientes mayor confianza y mejorar su calidad de vida [5].

La integración automatizada de información, desde estos sistemas, a las bases de datos de investigación puede proporcionar a la comunidad médica y científica la posibilidad de extracción inteligente de datos e información, para la mejora en la comprensión de las evoluciones de enfermedades.

1.3 Objetivos del Estudio

Uno de los mayores problemas de este tipo de redes es el consumo de energía, ya que para desarrollar su misión es necesario que permanezcan operativos durante largos periodos de tiempo.

El objetivo de este estudio se basa en intentar reducir dicho consumo, evitando que haya que cambiar las baterías de los nodos que forman la red cada poco tiempo.

Este proyecto pretende conseguir mejoras de consumo en estos nodos, integrando en ellos un algoritmo de diagnostico automático que realiza un análisis en tiempo real de los datos leídos por los sensores y reduce notablemente el numero de transmisiones necesarias del nodo a la estación base.

El algoritmo, del que se hablará más tarde en el *Apartado 2.2*, detecta ondas características del ECG de un individuo haciendo un análisis de los datos que ha recogido el sensor y evaluando si hubiera alguna anomalía u otra patología cardiaca.

El enfoque que se seguía hasta ahora era realizar el análisis del ECG en la estación base (PC, PDA, etc.), y solo encomendar al nodo la tarea de recoger los datos (pulsos eléctricos de la señal) y mandarlos a la estación. Ya que, debido al reducido tamaño, los sensores tienen grandes limitaciones de memoria y capacidad de cómputo.

Tras analizar los niveles de consumo, se decidió que fuera el nodo el encargado de realizar también la tarea de análisis y evaluación de los datos recogidos y no la estación base.

El motivo por el cual se detectan las ondas características de la señal ECG en el nodo, y no en la estación base (que tendría menos limitaciones de memoria), es que el envió de paquetes con los datos del sensor a la estación tiene un consumo de energía muy alto, ya que el elemento que más consume en el sensor es la radio, como se puede apreciar en la *Fig. 1.4*. Si el nodo solo recogiera lo datos, debería enviar todos a la estación base (para que esta pudiera realizar el análisis).

Sin embargo, si el sensor analiza los datos que ha recogido, y sólo manda los puntos característicos del ECG (detecciones) o las posibles patologías cardiacas tras evaluarla, se consigue reducir el número de transmisiones a la estación base.

Las modificaciones en el comportamiento del algoritmo original de diagnostico basado en ECG (para que solo se envié a la estación base los avisos la existencia de alguna cardiopatía en la detección si los hay, y variantes de dicho comportamiento, como solo enviar durante un periodo de tiempo, mientras otro periodo de tiempo se mantiene sin enviar nada, o solo enviar "avisos de la existencia de una patología" si estos se repiten varias veces consecutivas, etc.) podrían mejorar aún mas este consumo.

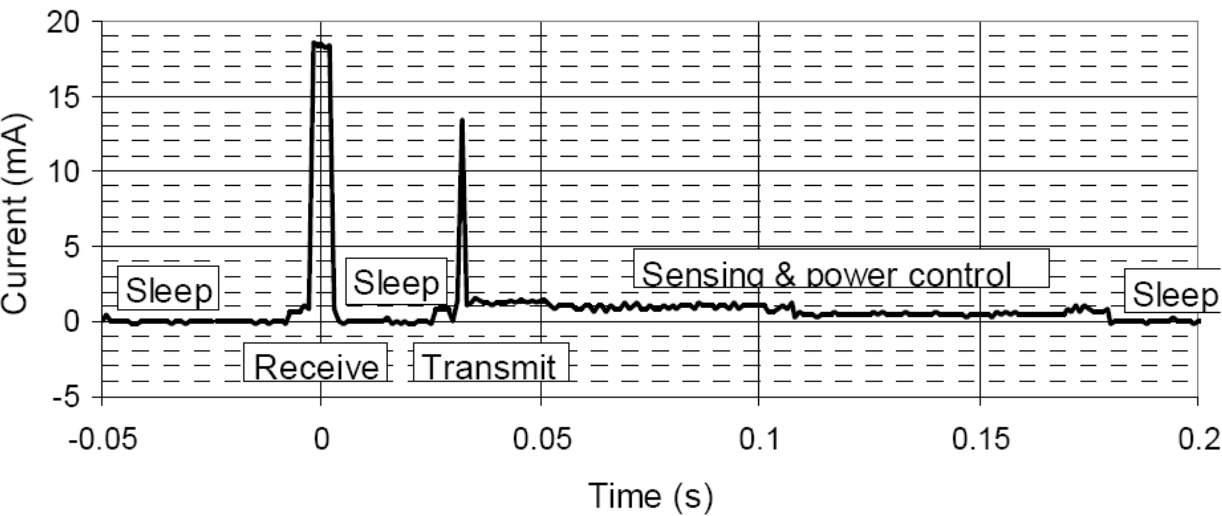


Figura 1.4: Estudio del perfil de Consumo de un nodo inalámbrico en sus diferentes fases.

En las siguientes secciones se explicará en que consiste el algoritmo de detección de ondas características del ECG, del que se ha partido en este proyecto, hasta llegar a las versiones finales que minimicen el consumo y se detallaran las características de los nodos corporales inalámbricos usados en la práctica.

2. Algoritmo Automático de diagnostico de ECG

2.1 Conceptos básicos sobre ECG y ondas características

El ECG es el registro de actividad eléctrica del corazón [21]. Las células cardiacas se despolarizan y se contraen. El ECG representa estas etapas de la estimulación y contracción del corazón.

Dentro de un electrocardiograma o ECG se distinguen un conjunto de ondas que caracterizan a este y nos aportan información sobre el estado de salud del individuo.

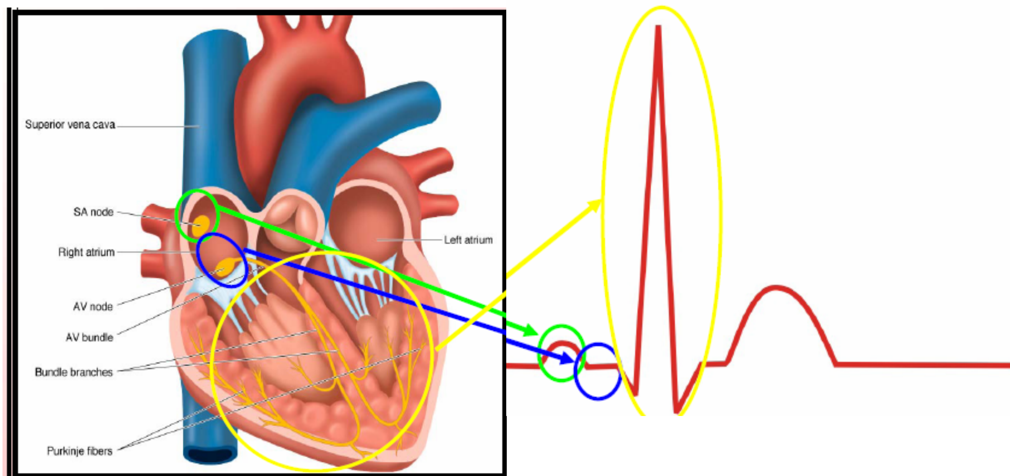


Figura 2.1: Forma característica de un ECG y las partes del corazón que las originan.

Las partes principales que se pueden distinguir de un ECG son las siguientes:

- Onda P: Indica la despolarización y contracción de las aurículas. Los puntos que marcan su inicio y final se denominan onset y offset de P

- Complejo QRS : Indica la despolarización y contracción ventricular

- Onda T: Indica la repolarización ventricular (la repolarización auricular queda enmascarada por el complejo QRS). Los puntos que marcan su inicio y final se denominan onset y offset de T

En el siguiente apartado, se describe un algoritmo que detecta las principales ondas características de un ECG.

2.2 Descripción del Algoritmo de Diagnostico y Detección de ECG

Basándonos en el algoritmo propuesto en el artículo de Yan Sun [1], que detecta ondas características de ECG usando una transformada morfológica y buscando en ella los puntos característicos eligiendo máximos y mínimos locales que superen unos umbrales, se ha implementado una versión inicial del algoritmo siguiendo estos pasos:

1) Preprocesado de la señal ECG usando filtrado morfológico (Morfological filtering - MMF).

   A la señal original se le aplican una serie de operaciones de apertura y cierre para reducir el ruido y la corrección de línea.

   $$Fb = Fo \circ Bo \bullet Bc$$

   $$F = \tfrac{1}{2} \left((Fo - Fb) \oplus B1 \ominus B2 + (Fo - Fb) \ominus B1 \oplus B2 \right)$$

   Donde,
   Fo es la señal original recogida por el sensor.
   Fb es la señal de corrección de línea.
   F es la señal después del preprocesado del paso 1.
   B1, B2, Bo y Bc son los elementos estructurales que se seleccionan basándose en las propiedades de las ondas características de ECG.

   Para el algoritmo, se han tomado Bo y Bc como dos segmentos horizontales de línea de amplitud cero cuyas longitudes vienen determinadas así:
   Lo= 0,2 * Frecuencia de muestreo (siendo la Frecuencia 200 Hz, Lo seria 40)
   Lc = 1,5 * Lo (siendo Lo = 40, Lc tendría un valor de 60)

   B1 y B2 se seleccionan según la Frecuencia de muestreo. En el algoritmo, son vectores de longitud 5 con los siguientes valores:
   B1 = {0, 1, 5, 1, 0} y B2 = {0, 0, 0, 0, 0}

   Las operaciones usadas en la corrección de línea son la apertura y el cierre, éstas se calculan a partir de la erosión y la dilatación:
   Apertura (o) $f \circ B = f \ominus B \oplus B$
   Cierre (●) $f \bullet B = f \oplus B \ominus B$

   Las operaciones morfológicas de erosión $\ominus$ y dilatación $\oplus$ se calculan de la siguiente manera:

   $$\text{Erosión: } (f \ominus B)(n) = \min_{m = 0, \ldots, M-1} \left\{ f\left(n - \frac{M-1}{2} + m\right) - B(m) \right\}$$

   $$\text{para } n = \frac{M-1}{2}, \ldots, N - \frac{M+1}{2}$$

$$\text{Dilatación: } (f \oplus B)(n) = \max_{m = 0, \ldots, M-1} \left\{ f\left(n - \frac{M-1}{2} + m\right) + B(m) \right\}$$

$$\text{para } n = \frac{M-1}{2}, \ldots, N - \frac{M+1}{2}$$

Para más información sobre los cálculos o ecuaciones de este punto, consultar el artículo de Yan Sun [3], donde se detalla con exactitud cada operación.

2) Transformada Morfológica Multiescala (Multiscale morfological transform - MMT).

En este paso, se obtiene una señal transformada, eligiendo una ventana de tamaño 2s+1 y buscando los valores máximos y mínimos, así como el valor central de la ventana.

Para hallar la transformada, se aplica la siguiente formula sobre los datos de la señal una vez preprocesados en el Paso 1.

$$M^{ds}{}_{f}(x) = \frac{\max\{f(t)\}\ t \in [x-s, x+s] + \min\{f(t)\}\ t \in [x-s, x+s] - 2 f(x)}{s}$$

Se debe cumplir la siguiente condición:

W (Anchura de la onda característica) * Fs (Frecuencia Muestreo) > s

La anchura del complejo QRS suele oscilar entre 0.06s y 0.12s, y las ondas P y T generalmente son mayores que dicho complejo. Por este motivo, en el artículo de Yan [1] se han elegido un valor W*F de 20 para la base de datos MITBIH de PhysioNet [4], y un valor de 15 para la base de datos QT de PhysioNet [4].

Figura 2.2: Transformada morfológica multiescala de un ECG

3) Detección de Máximos y Mínimos Locales. Selección de los Umbrales.

La selección de los umbrales *Thr* y *Thf*, que se usará posteriormente para determinar los puntos característicos del ECG, se basa en un método de *Thresholding* adaptativo sobre el histograma de la señal transformada del Paso 2.

4) Detección del pico característico del complejo QRS: el pico R.

Para detectar el pico de mayor amplitud (Rpeak), se busca el mínimo local en la onda Transformada del Paso2, cuya amplitud sea menor que el umbral -*Thr*.

5) Detección de la onda R (Rwave)

Por cada pico R que se haya detectado, se detecta su onda R, a partir de la posición en la que se detectó Rpeak y sobre la señal transformada del Paso 2.

El comienzo de la onda R será el primer máximo local hacia la izquierda, que supere el umbral *Thf.*

El final de la onda R se definirá como el primer máximo local, desde Rpeak hacia su derecha, que supere el umbral *Thf.*

6) Detección de la onda Q

En este paso se detecta el onset de la onda Q. Desde el comienzo de la onda R, hacia la izquierda, sobre la señal transformada del Paso 2, detectaremos como Q el primer mínimo local mayor que *Thf* (en valor absoluto).

Puede darse el caso de que no se encuentre Q, si tras 0,12 seg no se ha detectado su inicio.

7) Detección de la onda S

Este paso es equivalente al anterior. Se detecta el offset de la onda S, desde el final de la onda R, hacia la derecha, sobre la señal transformada del Paso 2, detectaremos como S, el primer mínimo local mayor que *Thf* (en valor absoluto).

Puede darse el caso de que no se encuentre S, si tras 0,12 seg no se ha detectado, al igual que para Q.

8) Detección del Onset, el pico y Offset de la onda P

La onda P precede al complejo QRS y sus extremos se denominan onset y offset de P respectivamente.

Para detectar esta onda, se buscan 2 máximos locales mayores que *Thf* y consecutivos (el onset y el offset de P) desde Q hacia la izquierda, en la onda transformada del Paso 2.

El mínimo local, que sea menor que -*Thf* (en valor absoluto), entre el onset y el offset de P en la señal transformada, será el pico de la onda P (P peak)

9) Detección del Onset, el pico y Offset de la onda T

La onda T se encuentra tras el complejo QRS y su inicio y fin se denominan onset y offset de T respectivamente.

Para detectar esta onda, se buscan 2 máximos locales mayores que *Thf* y consecutivos (el onset y el offset de T) desde S hacia la derecha, en la onda transformada del Paso 2.

El mínimo local, que sea menor que -*Thf*, entre el onset y el offset de T, en la señal transformada, será el pico de la onda T (T peak).

Tras ver los pasos fundamentales y el funcionamiento del algoritmo, se van a comentar a continuación, las modificaciones que se realizaron en el algoritmo para que pudiera funcionar, de forma completa y correcta, en el nodo. También se verán algunas de las variaciones para mejorar alguno de los puntos críticos y limitaciones del nodo (como la memoria y capacidad de procesamiento).

2.3 Evolución a un Algoritmo dinámico y optimizado

Hay ciertas diferencias entre la versión implementada del algoritmo y el algoritmo propuesto en el articulo de Yan Sun [1].

La diferencia más destacada es el **análisis y detección de los puntos de la señal en tiempo real** (como hace el algoritmo implementado), en contra de un análisis y detección de puntos característicos offline (como se propone en el artículo, en el que se cargaría la señal completa desde un fichero).

Hay varios motivos por los que se ha modificado el algoritmo propuesto, para hacer un análisis y detección dinámica, en vez de hacerlo sobre toda la señal, de forma estática. El más importante es el deseo de analizar una señal ECG de forma continuada durante un periodo de tiempo indefinido. Por otro lado, el nodo no puede almacenar gran cantidad de datos, así que memorizar y analizar la señal entera en el nodo es imposible.

Aún así, en el análisis de la señal, se usan operaciones (como las operaciones morfológicas para el filtrado de ruido, los métodos para hacer la transformada de la señal, etc.) que necesitan la señal entera o una parte de esta. Por este motivo, se va almacenando y analizando solo una pequeña parte de la señal, en un buffer circular de 300 valores. Se eligió esta longitud para que el buffer fuese lo suficientemente grande para almacenar un latido completo (onda P, complejo QRS y onda T).

Existen más modificaciones, que han sido requeridas, para poder usar el algoritmo en el nodo, con un correcto funcionamiento.

Debido a las limitaciones de procesamiento y memoria del sensor, algunas de las modificaciones realizadas al algoritmo han sido las siguientes:

- El MSP430 [9] del nodo no tiene unidad en punto flotante, por lo estas operaciones suponen un gran coste computacional. Por este motivo, todos los datos y operaciones en punto flotante han sido transformadas a enteros.

  Para realizar la transformada morfológica, se necesita una división, y al trabajar con datos enteros se necesita escalar la señal para disminuir la pérdida de precisión. Tras varias pruebas, se llegó a la conclusión de que basta con un decimal para conseguir una señal transformada válida para realizar las detecciones de las curvas características según el algoritmo.

- Definición de umbrales *Thr y Thf* estáticos, sin usar el método de Thresholding. En el algoritmo implementado, estos umbrales se han fijado, en vez de hallarse dinámicamente, a los valores *Thr* = 140 *Thf* = 5, debido a que no se obtuvieron buenos resultados usando el método de Thresholding que proponía el articulo. Se eligieron estos valores ya que se obtenían buenas detecciones, para todas las señales probadas, aunque es probable que se deban revisar, al probar el algoritmo con nuevas señales.

- Uso de señales ya filtradas, para el análisis y detección de ondas características (supresión del Paso 1), debido a que las operaciones morfológicas y los métodos de tratamiento de señales conllevan un gran uso de memoria y gran tiempo de procesamiento de datos, por lo que no se podía conseguir el funcionamiento adecuado. Se deberían estudiar más formas o métodos de filtrado de señales, que solucionaran este problema para el nodo.

- El nodo recibe datos a una frecuencia de 1000Hz, por lo que cada 0.001 segundos salta un evento (timer) que permite realizar operaciones con el dato recibido (enviarlo a la estación base o analizarlo). Para reducir dicha frecuencia a 200Hz, solo se toma un dato de cada 5. Al principio, se ejecutaba el algoritmo completo cada vez que se tenía un nuevo dato, pero la capacidad de procesamiento del nodo es insuficiente para realizar todas las operaciones antes de que reciba el siguiente dato. Por este motivo, el algoritmo esta dividido en 8 partes, que se detallan a continuación, realizando la que corresponda cada vez que se recibe un dato.

  1. Introduce un nuevo dato en el buffer circular y actualiza los contadores y punteros.
  2. Detección del pico R, si no lo encuentra salta de nuevo al paso 1. Si lleva varias veces sin encontrarlo, da error y pasa al paso 8.

3. Detección de la onda R. Si no lo encuentra salta al paso 1.
4. Detección del onset de Q.
5. Detección del offset de S.
6. Detección de la onda P.
7. Detección de la onda T. Si no la encuentra, deja tiempo para que entren más valores en el buffer, mientras onda P no se salga de este, si no dará error en el paso 8.
8. Validación de la detección y envío de resultados a la estación base.

Para el estudio de consumo y realizar mejoras sobre él, se partirá de una versión optimizada funcionando correctamente de este algoritmo.

En el *Apartado 4.1*, se repasarán los diferentes cambios y optimizaciones, hechas en dicha versión, que pretenden mejorar el consumo de energía, cuando el nodo esta detectando las ondas características.

Las señales que se han usado en las pruebas del algoritmo, se han extraído de la base de datos *Physionet* (http://www.physionet.org/cgi-bin/chart) (que guarda en su banco de datos muestras de señales biomédicas), en donde se encuentran muchos electrocardiogramas con múltiples patologías cardiacas.

La precisión del algoritmo ejecutándose en el nodo de 25 canales EEG/ECG se ha probado con varias señales tomadas de la base de datos QT de Physionet [4]. Esta base de datos consiste en 105 señales de 15 minutos elegidas de las otras bases de datos de Physionet.

El programa implementado para el nodo se ha comparado con el algoritmo de Yan Sun [1] original, un algoritmo basado en umbrales adaptativos (TD [29]) y un algoritmo basado en la transformada de wavelet (WD [30]), empleando los siguientes parámetros:

- La sensibilidad (Se)**,**

$$Se = \frac{TP \times 100}{TP + FN}$$

Donde: TP es el número de detecciones verdaderas
FN es el número de detecciones anotadas manualmente y que no han sido detectadas por el algoritmo.

- El error medio (m),
- La desviación estándar (σ)

El comité de Common Standards for Electrocardiography (CSE), establece unos valores tolerables para σ.

Algoritmo	Parametos	Pon	Poff	QRSon	QRSoff	Ton	Toff
Adaptacion	Se (%)	92.4	92.4	100	100	96.6	91.7
	m (ms)	1.4	14.9	-7.8	8.2	53.6	12.8
	σ (ms)	15.6	13.3	22.6	16.8	21.6	20.9
MMD	Se (%)	97.2	94.8	100	100	99.8	99.6
	m (ms)	9.0	12.8	3.5	2.4	7.9	8.3
	σ (ms)	9.4	13.2	6.1	10.3	15.8	12.4
TD	Se (%)	96.2	97	99.9	99.9	98.8	98.9
	m (ms)	10.3	-5.7	-7.3	-3.6	23.3	18.7
	σ (ms)	14.1	13.6	10.9	10.7	28.3	29.3
WD	Se (%)	89.9	89.9	100	100	99.1	99.2
	m (ms)	13	5.4	4.5	0.8	-4.8	-8.9
	σ (ms)	12.7	11.9	7.7	8.7	13.5	18.8
CSE	σ (ms)	10.2	10.7	6.5	11.6	-	30.6

Tabla 2.1. Comparación de los la exactitud de los diferentes algoritmos

Como se ve en la tabla 2.1, los resultados obtenidos no son tan buenos como para el algoritmo MMD original, pero esto se debe a la pérdida de precisión asociada al uso de datos enteros en vez de punto flotante. Aún así, los valores no se alejan demasiado de los obtenidos en el algoritmo original.

Las imágenes 2.3, 2.4 y 2.5 muestran algunas señales ECG con los puntos característicos señalados, detectados mediante el algoritmo modificado. Como se ve el algoritmo detecta todos los puntos correctamente.

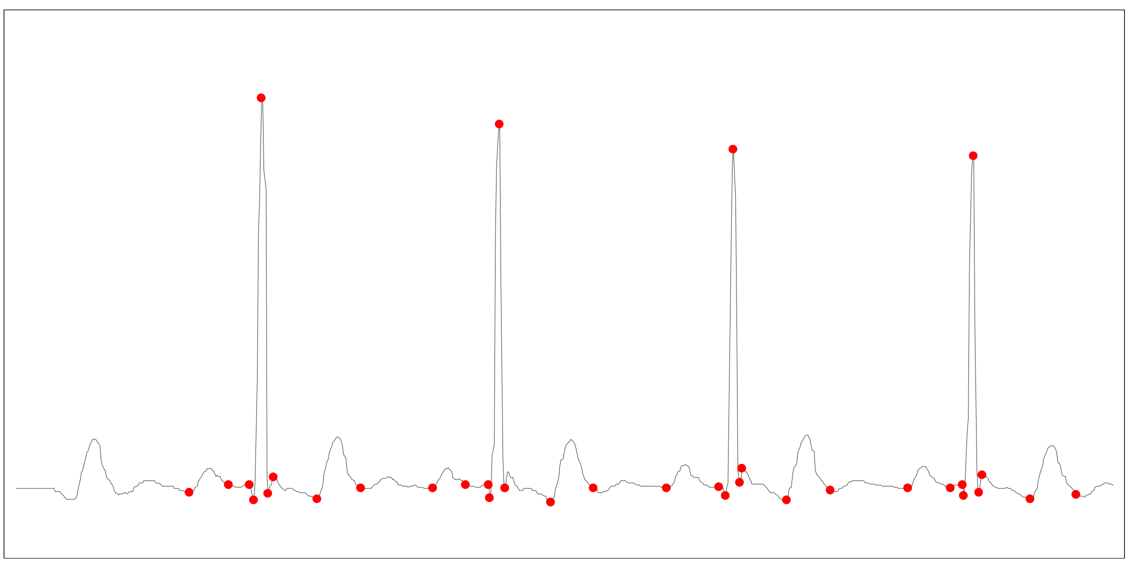


Figura 2.3: Detección de las ondas características en un ECG usando el algoritmo adaptado al el nodo

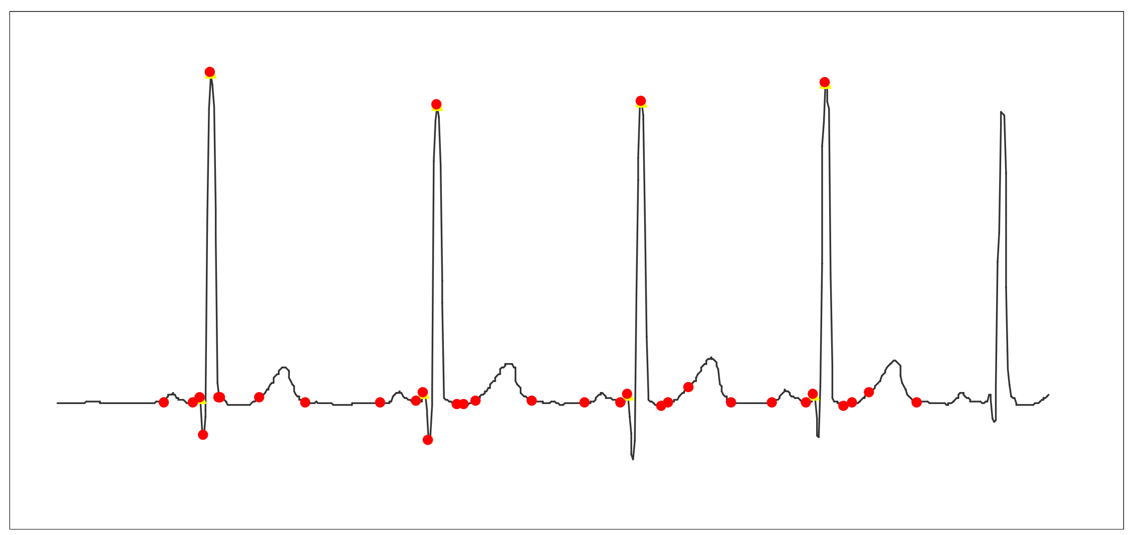


Figura 2.4: Detección de las ondas características en un ECG usando el algoritmo adaptado al el nodo

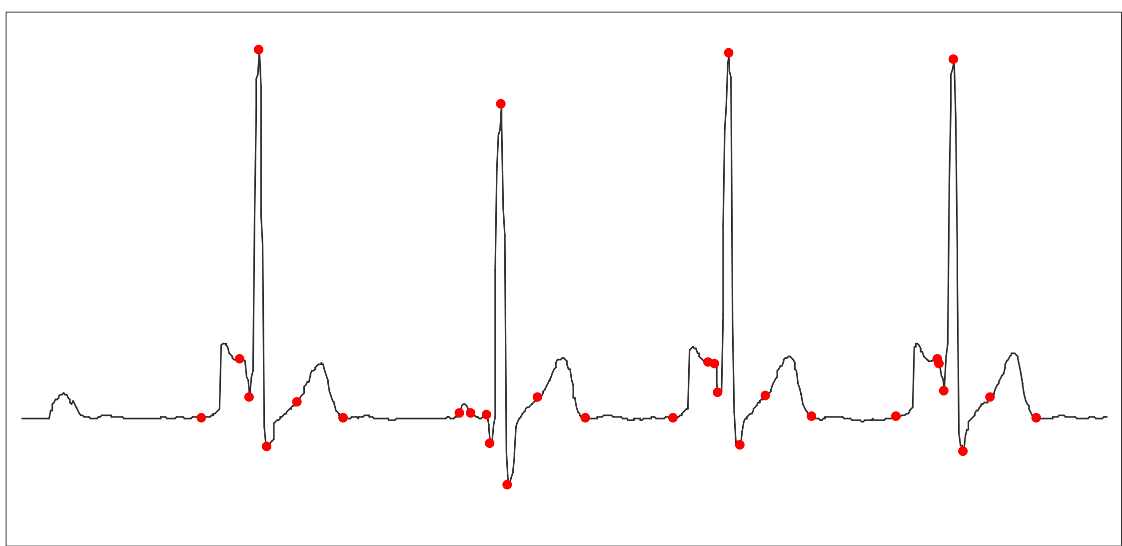


Figura 2.5: Detección de las ondas características en un ECG usando el algoritmo adaptado al el nodo

2.4 Algoritmo de diagnóstico

En la figura 2.6 se distinguen varios de los puntos característicos que va a detecta el algoritmo descrito en el apartado anterior.

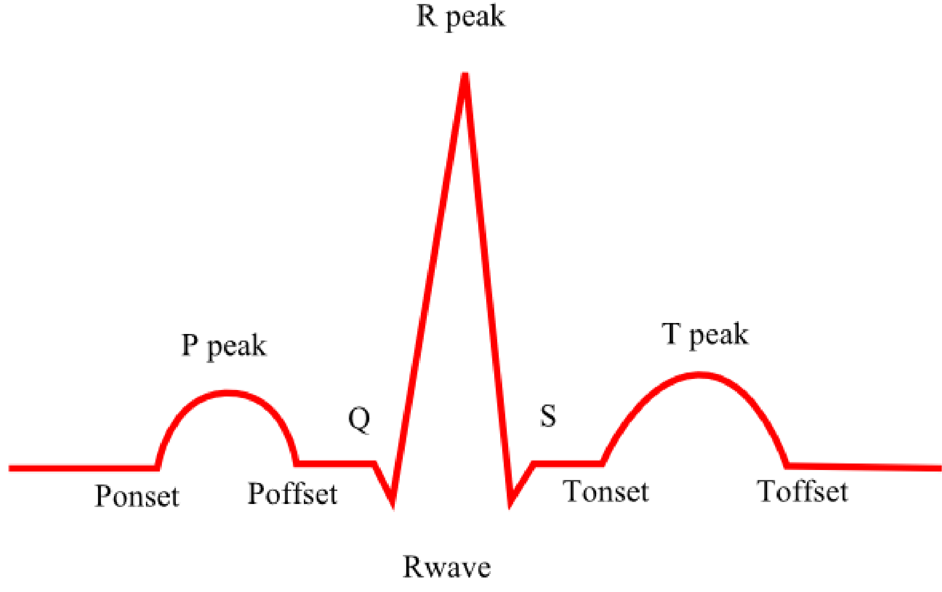


Figura. 2.6: Complejo QRS y ondas P y T

Teniendo en cuenta los valores normales de la frecuencia cardiaca (tabla 2.2) y aplicando una serie de reglas que se deberían cumplir en un individuo sano podemos detectar anomalías asociadas a problemas cardiacos comunes.

La frecuencia cardiaca es el número de latidos por unidad de tiempo, y normalmente se expresa en latidos por minuto. Depende de muchos factores pero hay unos rangos establecidos para los adultos según su condición y la actividad física que estén realizando.

	Sedentario	**En forma**	**Deportista**
En reposo	70 - 90	60 – 80	40 – 60
Aeróbico	110 – 130	120 – 140	140 – 160
Anaeróbico	130 – 150	140 – 160	160 – 200

Tabla. 2.2: Frecuencias Cardiacas normales.

Reglas de Normalidad:

1) La distancia desde Q a S debe de ser ≤ 0,10 seg.

2) La distancia de P onset a Q debe de ser estar comprendida entre 0,2 y 0,12 seg.

3) T peak siempre debe de ser positivo.

4) La distancia entre Q y Rpeak no debe ser mayor de 0,03 seg.

5) El intervalo QT es el intervalo que se mide desde Q a T offset.
Usando una corrección de éste respecto a la frecuencia cardiaca con la fórmula de Bazet, se obtendría el QTc:

$$QTc = \frac{time\ interval\ from\ D\ to\ K}{\sqrt{previous\ RR\ interval}}$$

Siendo RR la distancia en segundos entre el pico R (Rpeak) de la detección actual y el Rpeak de la anterior detección.

La corrección del intervalo QT (QTc) debe estar comprendida dentro de los valores normales que se muestran en la tabla de la *Fig. 2.7*

Frecuencia cardiaca/min	Intervalo RR (s)	QTc (s) y límites normales
40	1,5	O,46 (0,41-0,51)
50	1,2	0,42 (0,38-0,46)
60	1	0,39 (0,35-0,43)
70	0,86	0,37 (0,33-0,41)
80	0,75	0,35 (0,32-0,39)
90	0,67	0,33 (0,30-0,36)
100	0,60	0,31 (0,28-0,34)
120	0,50	0,29 (0,26-0,32)
150	0,40	0,25 (0,23-0,28)
180	0,33	0,23 (0,21-0,25)
200	0,30	0,22 (0,20-0,24)

Figura. 2.7 Criterios de normalidad del ECG dependiendo del intervalo RR anterior y de la corrección del intervalo QT

En los casos en los que no se hayan detectado los puntos Q y S, se aplicarán las mismas reglas para validar las detecciones, sustituyéndolos por el inicio y final de la onda R respectivamente.

Es posible distinguir determinadas enfermedades o patologías basándonos en estas reglas de normalidad. En la tabla 2.3 podemos ver los problemas más comunes asociados con el incumplimiento de las reglas anteriores.

Para comprender las explicaciones de las posibles patologías de la tabla se requiere un conocimiento avanzado de la materia.

	Posible Problema Cardiaco
Regla 1	Bloqueo del Haz de His. Ritmo supraventricular con conducción anormal.
Regla 2 (> 0.2 seg)	Desorden en la conducción entre aurículas y ventrículos a un nivel de nodo auroventricular, Haz de His o el sistema de Purkinje
Regla 2 (< 0.12 sg)	Presencia de un camino de acceso anómalo, que origina una conducción mas rápida o la presencia de un ritmo con origen en la unión auroventricular, en la aurícula izquierda o en la parte inferior de la aurícula derecha. Generalmente, esta anomalía es debida a una pre-excitación ventricular.
Regla 3	Alteraciones primarias de la fase de repolarización (por isquemia o infarto de miocardio, pericarditis o miocarditis) Alteraciones secundarias de la fase de repolarización (por alteraciones en la repolarización ventricular)
Regla 4	Retraso en el tiempo de activación ventricular
Regla 5 (si es mayor)	La repolarización ventricular se ha ralentizado por causas adquiridas o congénitas. Relacionado con la aparición de arritmias.
Regla 5 (si es menor)	Problema normalmente relacionado con el uso de algunas medicinas, hipercalcemia o hiperpotasemia.

Tabla 2.3. Posibles cardiopatías relacionadas con el incumplimiento de las reglas de normalidad definidas en este apartado

Tras la detección de las ondas características, se añade un método en el algoritmo que valida las detecciones que ha encontrado, viendo si se cumplen estas Reglas de Normalidad.

Este método, además de hacer más completo el algoritmo para poder detectar de forma más concreta algunas anomalías en las detecciones, pretende mejorar el consumo en el sensor, ya que se podrá distinguir un problema directamente en el nodo (sin tener que validar la detección en la estación base) y sólo enviar la información de un latido en caso de que este se salga de la normalidad y su fallo correspondiente. Evitando así un consumo de energía innecesario.

Por cada latido que se detecta, se el algoritmo devuelve el resultado de la validación, el instante de tiempo en el que detecto el pico R, y la posición de los demás puntos característicos (inicio y final de R, Q, S, onset y offset de P, onset y offset de T), respecto de dicho instante. Así como las amplitudes de las ondas R, P y T.

De forma un poco más gráfica:

Resul (8 bits)	Ho (8)	Min (8)	Sg (8)	Milisg (8)	Rw0 (8)	Rw1 (8)	Q (8)	S (8)	onsetP (8)	offsetP (8)	onsetT (8)	offsetT (8)

RA (16 bits)	PA (16)	TA (16)

Resul: Resultado que depende de la validación de la detección y las Reglas de Normalidad. En la *tabla 2.4* se pueden ver los diferentes valores que puede tomar "Resul" y sus traducciones correspondientes a posibles patologías.
Ho: hora en la que hay un Rpeak.

Min: minuto en la que se encuentra el Rpeak dentro de esa hora “Ho”.
Sg: segundo en el que está el Rpeak de ese minuto “Min”.
Milisg: milisegundo concreto del Rpeak en ese segundo “Sg”.
Rw0: inicio de la onda R respecto de Rpeak
Rw1: final de la onda R respecto de Rpeak.
Q: inicio de la onda Q respecto de Rpeak
S: final de la onda S respecto de Rpeak
OnsetP y offsetP: inicio y final de la onda P respecto de Rpeak
OnsetT y offsetT: inicio y final de la onda T respecto de Rpeak
RA: Amplitud de la onda R (valor de Rpeak en la señal tras el filtro de ruido).
PA: Amplitud de la onda P (valor de Ppeak en la señal tras el filtro de ruido).
TA: Amplitud de la onda T (valor de Tpeak en la señal tras el filtro de ruido).

Resultado	Estado del ECG (basándose en la validación y las detecciones)
0	Todo correcto. Se detectaron todos los puntos característicos y la detección cumple todas las reglas de Normalidad.
1	Falla la regla de normalidad 1 para los valores de la detección.
2	Falla la regla de normalidad 2 para los valores de la detección.
3	Falla la regla de normalidad 3 para los valores de la detección.
4	Falla la regla de normalidad 4 para los valores de la detección.
5	Falla la regla de normalidad 5 para los valores de la detección
8	No se encontró onda P al hacer las detecciones.
9	No se encontró onda T al hacer las detecciones.
10	No se ha detectado el pico R (Rpeak) en un cierto tiempo (< 3 seg.)

Tabla 2.4. Valores que puede tomar el Resultado de la detección tras analizar la señal con el algoritmo y su significado en relación a la búsqueda de ondas características y su validación.

3 Arquitectura de los nodos

El nodo utilizado para realizar este proyecto, tiene las siguientes características:

- Consta de 25 canales que pueden monitorizar señales EEG (encefalograma) y ECG (electrocardiograma).
- Pueden transmitir en tiempo real la información medida a la estación base.
- Tienen una arquitectura software muy completa, *Fig.3.1*, permitiendo integrar aplicaciones de procesado de señales, usar diferentes componentes hardware y manejar la pila de comunicación para varios protocolos de comunicación inalámbricos.

Application		
MAC	TinyOS Kernel	
Radio driver	uC driver	ASIC driver
Radio nRF2401	uC MSP430	25-ch ASIC

Figura. 3.1: Arquitectura del nodo con un sensor basado en EEG/ECG 24+1 canales.

El sistema completo necesita un aporte de energía de entre 2.7V a 3.3V (como 2 pilas alcalinas AA) como fuente de alimentación.

Otra posibilidad, es el uso de *energy scavengers*, que son dispositivos que transforman la energía ambiental a energía eléctrica. Hay varios tipos, pero destacan los que son capaces de transformar el movimiento en energía. También hay otro tipo que consigue transformar en electricidad las diferencias de temperatura.

A continuación se explican con más detalle las características HW y SW de estos nodos.

3.1 Arquitectura Hardware

El nodo está formado por tres componentes principales: microcontrolador, sensor y Radio.

El sensor es un ASIC de muy bajo consumo y 25 canales capaz de captar actividad eléctrica de señales EEG y ECG. La tasa de muestreo máxima es 1024Hz por canal.

El microcontrolador trata de un dispositivo MSP430x149 [9] de bajo consumo de Texas Instruments diseñado para dispositivos empotrados, especialmente debido a su bajo consumo, con 60kB de memoria flash (ROM) y 2kB de RAM.

Para comunicarse, el nodo utiliza una Radio modelo Nordic nRF2401 [8], un chip de bajo consumo, con un ancho de banda de 2.4 a 2.5GHz.

3.1.1 MSP430

El microcontrolador TI MSP430 [9] consiste en un procesador de 16bits con arquitectura RISC, algunos periféricos y un sistema de reloj flexible, interconectados usando un bus común para datos y memoria (con arquitectura von-Neumann) (imagen 3.2).

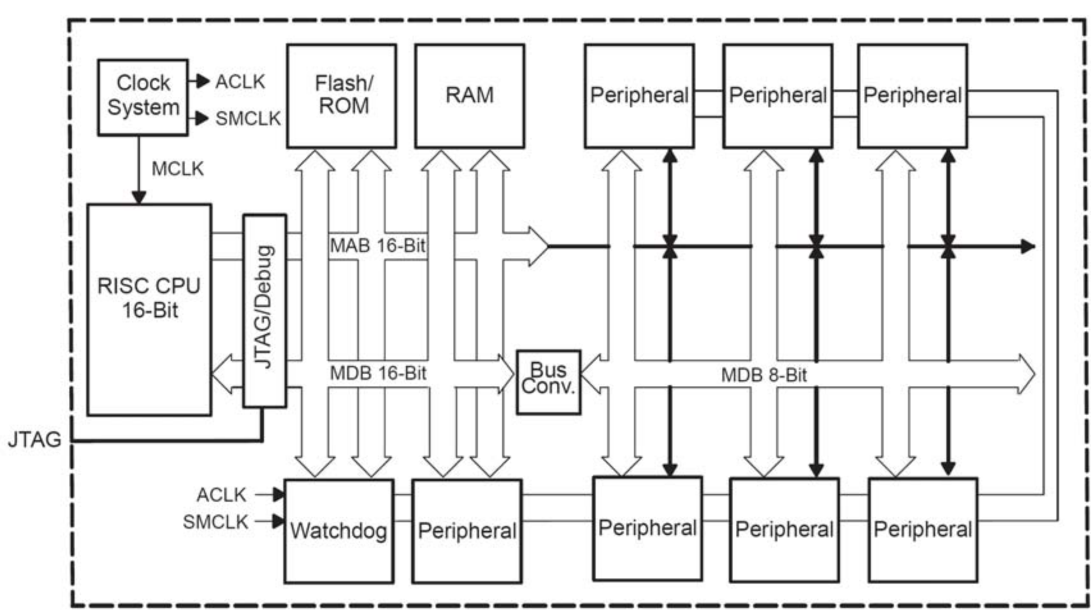


Figura. 3.2: Arquitectura del microcontrolador TI MSP430

También tiene de una serie de periféricos, como un conversor analógico-digital de 12 bits, que es especialmente útil para el sensor EEG/ECG ya que facilita el procesamiento de señales biomédicas.

Esta arquitectura RISC se caracteriza por:

- Tener un juego de 27 instrucciones y 7 modos de direccionamiento.
- Ser ortogonal, es decir, cualquier instrucción puede usar cualquiera de los modos de direccionamiento.
- Incorporar 16 registros de 16 bits totalmente accesibles, que tienen funciones específicas (R0, R1, R2 y R3) o son de propósito general (de R4 a R15).

La memoria consiste en único espacio de direcciones compartido. Con 60kB destinadas a memoria ROM y 2kB de RAM. La ROM puede almacenar tanto código como datos usándose sin necesidad de copiarse en la RAM.

El reloj de sistema esta diseñado para dispositivos de bajo consumo, ya que permite varios modos de operación configurables por software, por lo que consta de dos relojes principales y uno secundario:

- ACLK: un reloj principal de baja frecuencia auxiliar (para el modo de bajo consumo stand-by).
- MCLK: un reloj principal de alta velocidad, para alto rendimiento.
- SMCLK: un reloj su-principal destinado a los periféricos.

El oscilador controlado digitalmente (DCO) del sistema, permite pasar de un estado o modo de operación a otro en menos de 6µs, teniendo en cuenta las necesidades del sistema. Por ejemplo, si llega una interrupción, puede pasar de cualquiera de los modos de bajo consumo a un modo activo (para resolver la interrupción) y volver al modo de bajo consumo en el que estaba.

Tiene un modo activo y 5 modos de bajo consumo:

Modo activo (AM): La CPU y todos los relojes están activos. Con un consumo bastante reducido de 0.6 nJ/instrucción.

Modo de Bajo consumo 0 (LPM0): La CPU esta desconectada. Los relojes ACLK y SMCLK están activos y MCLK desconectado.

Modo de Bajo consumo 1 (LPM1): La CPU esta desconectada. Los relojes ACLK y SMCLK están activos y MCLK desconectado. El generador DC estará desconectado si el DCO no se usa en el modo activo.

Modo de Bajo consumo 2 (LPM2): La CPU esta desconectada. ACLK está activo y SMCLK y MCLK desconectados.

Modo de Bajo consumo 3 (LPM3): La CPU esta desconectada. ACLK está activo y SMCLK y MCLK desconectados. El generador DC estará desconectado.

Modo de Bajo consumo 4 (LPM4): La CPU y todos los relojes están desconectados.

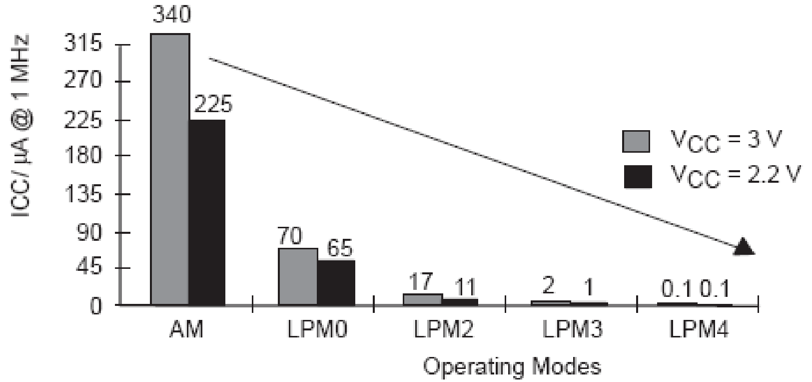


Figura 3.3: Consumo en los diferentes modos de operación en el MSP430

En la *Fig. 3.3,* se puede observar el decremento del consumo en el microcontrolador usando cualquiera de los modos de bajo consumo. En estado activo, funcionando a 1MHz y con un voltaje de 2.2V tiene un consumo de 280µA, mientras que en el modo de bajo consumo 4 tendrá 0.1µA.

Si aumenta la frecuencia de reloj, aumenta la energía necesaria para operar y por tanto el consumo de energía. El máximo consumo se alcanza en el modo activo, con el reloj funcionando a una frecuencia de 8MHz y un voltaje de 3.6V (*Fig. 3.4*)

Para más información del TI MSP430 consultar las referencias [9].

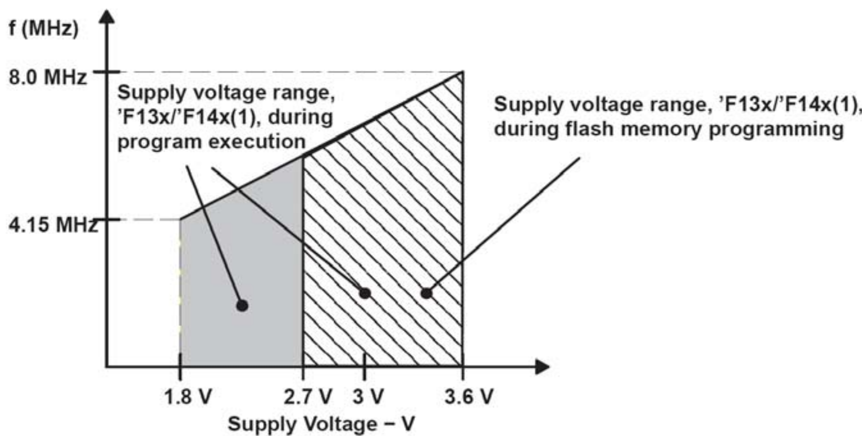


Fig. 3.4: Frecuencia vs. Vin del MSP430

3.1.2 Nordic nRF2401

Este dispositivo [8] esta formado por un sintetizador de frecuencia integrado, un amplificador de potencia, un oscilador de cristal y un modulador, para permitir la comunicación inalámbrica.

Para configurar la radio, se carga una palabra de configuración (15 bytes) mediante una interfaz de 3 conexiones (CE, CLK1, DATA). Esta palabra decide la funcionalidad del dispositivo en cada modo de configuración y la tasa de datos se decide por la velocidad del microcontrolador asociado.

El consumo de este modelo de radio nRF2401 es muy bajo en comparación con otros modelos, solo de 10.5mA en modo de transmisión (con una potencia de salida de -5dBm) y 18mA en el modo receptor para un voltaje de entrada de 1.9 a 3.6 V. Sin embargo, aunque tiene cinco modos de consumo diferente, es el componente del nodo con las mayores tasas de consumo.

El nRF2401, puede establecerse en los distintos modos de consumo principales, dependiendo de los 3 pines de control (PWR_UP, CE y CS) como se muestra en la *Tabla 3.1*:

Mode	PWR_UP	CE	CS
Active (RX/TX)	1	1	0
Configuration	1	0	1
Stand by	1	0	0
Power down	0	X	X

Tabla 3.1: Principales modos de consumo del nRF2401.

El modo ShockBurst y el modo Direct Mode son los dos modos activos de la Radio. Vamos a detallar las características de estos 4 modos de consumo:

1. **Modo ShockBurst (modo activo)**:

   Cuando se opera en este modo, la tasa de datos llega a un máximo de 1Mbps con un ancho de banda de 2.4GHz, No se requiere al microcontrolador para el procesamiento de datos. Por lo que consigue una gran velocidad de transferencia con un bajo rendimiento del procesador. Este modo reduce el consumo del dispositivo considerablemente.

   La importancia de este modo de comunicación radica en que el microcontrolador puede enviar datos para ser enviados a una frecuencia diferente a la que la radio envía dichos paquetes (figura 3.5).

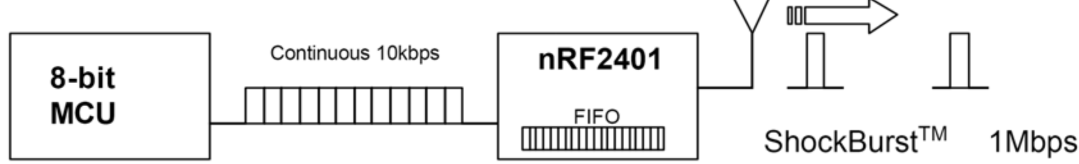


Figura 3.5: No es necesaria la sincronización de los datos entre el microcontrolador y la radio

Para conseguirlo, el transmisor consta de un buffer FIFO que almacena los datos que le llegan del procesador, y que solo envía una vez que está lleno a la velocidad que estime (0 – 1Mbps). De esta forma, la radio puede estar enviando a máxima frecuencia, mientras que el procesador esta en un modo de bajo consumo (figura 3.6).

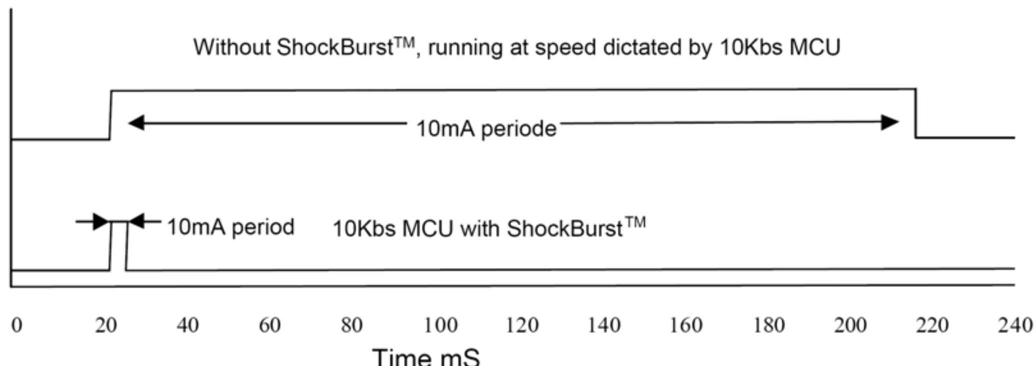


Figura 3.6: Consumo sin ShockBurst vs. Consumo con ShockBurst

Transmisión en modo ShockBurst (TX):
Cuando el microcontrolador tiene un dato para enviar, activa la señal CE, esto configura el NRF2401 para procesado de datos.

Se comienza a rellenar el paquete fijando la dirección del nodo destino y los datos. El protocolo de la aplicación o el microcontrolador establecen la velocidad de transmisión entre 0 y un máximo de 1Mbps.

El microcontrolador establece CE a cero, y esto activa la transmisión ShockBurst, que conlleva los siguientes pasos:

1. Se enciende la radio.
2. Se completa el paquete (Se añade la cabecera y se calcula el CRC).
3. Se transmiten los datos a alta velocidad (250 kbps o 1 Mbps).
4. El transmisor vuelve al modo stand-by cuando termina.

Recepción en modo ShockBurst (RX):

Cuando se establece el modo de recepción (RX) de ShockBurst al activar la señal CE, se determina la dirección y el tamaño de los datos de los paquetes de entrada.
Cuando se recibe un paquete válido (dirección y CRC correctos) se queda solo con el campo de datos y se notifica al microcontrolador (confirmación de envió), estableciendo el pin DR1 a alta.
Se podría deshabilitar la radio (Power Down Mode), si el microcontrolador desactiva la señal CE. Este también fija los datos en una tasa adecuada (como 10Kbps).

Cuando se han recibido todos los datos, se deshabilita la señal DR1 y entonces, estará preparado para la llegada de nuevos paquetes si el CE se ha dejado a alta. Si no, se repite el proceso de recepción.

2. **Direct Mode (modo activo):**

El nRF2401 funciona como un receptor tradicional, la tasa de bits debe ser de 1Mbps ±200ppm (o 250Kbps para una configuración de tasa de datos baja), para que el receptor detecte las señales.

Transmisión en modo Directo:
Cuando el microcontrolador tiene un dato, activa CE. La radio se activa inmediatamente y tras 200us, los datos se cargan directamente.
Todas las partes del protocolo de radio deben estar implementadas en firmware para definirlas por el microcontrolador (preámbulo, dirección y CRC).

Recepción en modo Directo:
Una vez que el nRF2401 ha sido configurado y encendido en modo de recepción (CE a alta), DATA empezara a oscilar debido a ruido. Al igual que el reloj, ya que el transmisor esta intentando bloquear el flujo de datos entrantes.
Una vez que llega una cabecera (preámbulo) válido, CLK1 y Data se bloquearán. El paquete llegará al puerto DATA con la misma velocidad que fue transmitido.
Para activar el demodulador y regenerar el reloj, el preámbulo debe ser de 8 bits de 1 y 0 alternos, comenzando con 0 si el primer bit del área de datos es 0.
En este modo, se dispone de la señal DR (no data Reddy). Y las direcciones y el CRC también deben establecerse en el microcontrolador.

3. **Stand-By Mode (Modo Bajo Consumo):**

Se usa para minimizar el consumo medio, mientras se mantienen tiempos de inicio cortos. El consumo medio depende de la frecuencia del oscilador (12µA con 4MHz, 32µA con 16 MHz). El contenido de la palabra de configuración se mantiene en este modo.

4. **Power Down Mode (Modo Bajo Consumo):**

Se desconecta el transmisor (consumiendo el mínimo posible, menos de 1µA), aumentando el tiempo de vida de la batería. El contenido de la palabra de configuración se mantiene en este modo.

3.1.3 La estación base

La estación base o receptor de datos, que se encarga recopilar la información que consiguen o analizan los nodos, tiene la misma arquitectura que los nodos normales pero en lugar de sensores tiene un conector USB para poder ser conectada a un ordenador. En un futuro, también podría ser un móvil, una PDA, etc.

El nodo envía a la estación base un paquete de 33bytes con la siguiente estructura.

Preámbulo	ADDR	PAYLOAD	CRC

Siendo:

- ADDR: dirección del destino.
- PAYLOAD: datos (18bytes).
- CRC: comprobación de redundancia cíclica (para errores).

3.2 Arquitectura Software

La arquitectura software de los nodos se divide en 3 módulos principales (figura 3.7), cada uno de los cuales se corresponde con los 3 bloques hardware más importantes. Esto permite modificar el comportamiento de dichos módulos o los elementos hardware, sin alterar el resto de módulos del sistema.

Application		
MAC	TinyOS Kernel	
Radio driver	uC driver	ASIC driver
Radio nRF2401	uC MSP430	25-ch ASIC

Figura. 3.7: Arquitectura del nodo con un sensor basado en EEG/ECG 24+1 canales.

Existen varios sistemas operativos desarrollados para nodos de este tipo. Se describen algunos de ellos a continuación:

CORMOS (Communication Oriented Runtime System for Sensor Networks) [15]
Específico para redes de sensores inalámbricas como su nombre indica.

Bertha (pushpin computing platform) [12]: Una plataforma de software diseñada e implementada para modelar, testear y desplegar una red de sensores distribuida de muchos nodos idénticos. Sus principales funciones se dividen en:

- Administración de procesos
- Manejo las estructuras de datos
- Organización de los vecinos
- Interfaz de Red

Nut/OS [13]: Es un pequeño sistema operativo para aplicaciones en tiempo real, que trabaja con CPUs de 8 bits. Está diseñado para procesadores con velocidad de 1 MIPS CPU, 0.5 kBytes RAM y 8 kBytes ROM. Tiene las siguientes funciones:

- Multihilo
- Mecanismos de sincronización
- Administración de memoria dinámica
- Temporizadores asíncronos
- Puertos serie de Entrada/Salida

Contiki[14]: Es un Sistema Operativo de libre distribución para usar en un limitado tipo de dispositivos, desde los 8 bits a sistemas empotrados en microcontroladores, incluidos nodos de redes inalámbricas.

eCos (embedded Configurable operating system) [16]: es un sistema operativo de libre distribución, en tiempo real, diseñado para aplicaciones y sistemas empotrados que sólo necesitan un proceso. Se pueden configurar muchas opciones y puede ser personalizado para cumplir cualquier requisito, ofreciendo la mejor ejecución en tiempo real y minimizando las necesidades de hardware.

EYESOS [17]: se define como un entorno para escritorio basado en Web, permite monitorizar y acceder a un sistema remoto mediante un sencillo buscador.

MagnetOS [18]: es un sistema operativo distribuido para redes de sensores o adhoc, cuyo objetivo es ejecutar aplicaciones de red que requieran bajo consumo de energía, adaptativas y fáciles de implementar.

t-Kernel [19]: es un sistema operativo que acepta las aplicaciones como imágenes de ejecutables en instrucciones básicas. Por ello, no importará si está escrito en
C++ o lenguaje ensamblador.

LiteOS [20]: Sistema operativo desarrollado en principio para calculadoras, pero que ha sido también utilizado para redes de sensores.

El nodo tiene un sencillo sistema operativo, basado en eventos, llamado TinyOS [22] especialmente diseñado para redes de sensores inalámbricas. Utiliza un lenguaje de programación llamado NesC [6], que es una extensión de C.

Gracias a esto, es posible portar aplicaciones usando los drivers provistos por TinyOs para acceder a los diferentes bloques hardware. Además, la abstracción de los bloques hardware hace posible modificarlos o reemplazarlos, sin perjudicar al resto del sistema.

A continuación se detallara las características principales de este sistema operativo, y su simulador, TOSSIM [23] así como la extensión de este para el estudio de consumo, PowerTossim .

3.2.1 Sistema Operativo TinyOS

El sistema operativo TinyOS [22] es un reducido núcleo multitarea, útil para pequeños dispositivos, como los nodos.

Este SO, de código abierto, funciona a partir de eventos producidos que llamarán a funciones. Ha sido desarrollado en la universidad de Berkeley, para redes de sensores con recursos limitados.

Tanto el sistema TinyOS, como sus librerías y aplicaciones, están implementadas en NesC [6] (lenguaje en el que hay que escribir las aplicaciones que se ejecutarán en los nodos con dicho sistema operativo), una versión de C que fue diseñada para programar sistemas empotrados, para más información sobre este lenguaje, consultar el *Manual de NesC* del apéndice II.

TinyOS tiene las siguientes principales características:

- El kernel ocupa 400 bytes entre código y datos.
- Arquitectura basada en componentes.
- Capas de abstracción bien establecidas, limitadas claramente a nivel de interfaces, a la vez que se pueden representar los componentes automáticamente a través de diagramas.
- Amplios recursos para elaborar aplicaciones.
- Adaptado a los recursos limitados de los nodos: energía, procesamiento, almacenamiento y ancho de banda.
- Operaciones divididas en fases (Split-phase).
- Dirigido por eventos (Event Driven).
- Concurrencia de tareas y basada en eventos.
- Implementación en Nesc.
- Una aplicación consiste en una configuración de alto nivel y todos los módulos asociados

El diseño de TinyOS se ajusta a las características y necesidades de las redes de sensores (reducido tamaño de memoria, bajo consumo de energía, operaciones de concurrencia intensiva, diversidad en diseños y usos, etc.). Y se encuentra optimizado en términos de uso de memoria y eficiencia de energía.

El diseño de su núcleo está basado en una estructura de dos niveles de planificación:

- Eventos: Son procesos asociados con eventos HW. Son rápidamente ejecutables y pueden interrumpir las tareas que se estén ejecutando. Cuando llega un evento, se guarda el estado actual del sistema, y cuando se completa, restablece el estado se continúa con la tarea que estuviera en curso cuando llegó.

Un sistema basado en eventos fuerza a las aplicaciones a declarar implícitamente cuando se acaba de usar la CPU. Ésta entra entonces en un estado de inactividad consumiendo el mínimo de energía.

- Tareas: Son contextos de ejecución que corre en background hasta completarse en su totalidad, sin inferir en otros eventos o tareas del sistema (exclusión mutua). Las tareas intentan hacer una cantidad mayor de procesamiento y no son críticas en tiempo, por lo que pueden ser interrumpidas por eventos.

Con este diseño se permite que los eventos (que son rápidamente ejecutables), puedan ser realizados inmediatamente, pudiendo interrumpir a las tareas. Y por tanto, alcanzar un alto rendimiento en aplicaciones de concurrencia intensiva. Además, este enfoque usa las capacidades de la CPU de manera eficiente y para consumir el mínimo de energía.

Este SO tiene un modelo de programación basado en componentes, es decir, que se encuentra construido sobre un conjunto de componentes, que son la base para la creación de aplicaciones (las aplicaciones serán una lista de componentes y la especificación de las interconexiones entre ellas). Con esto, se permite la fácil migración a otro hardware, dada la abstracción que se logra con el modelo de manejo de eventos.

Cada componente consta de:

- Manejador de comandos: Los comandos son peticiones hechas a componentes de capas inferiores. Estos generalmente son solicitados para ejecutar alguna operación.

- Manejador de eventos: Los manejadores de eventos son invocados por eventos de componentes de capas inferiores, o por interrupciones cuando se está directamente conectado al hardware.

- Un *frame* de tamaño de datos privado, con asignación estática de memoria.

- Un bloque con tareas simples: Las tareas son entregadas a un planificador (*task scheduler*) que en este caso está implementado con método FIFO (las tareas se ejecutan secuencialmente).

Se distinguen tres tipos de componentes según su nivel de abstracción:

- Abstracciones de Hardware: Mapean el HW físico en el modelo de componentes.
- Hardware Sintético: simulan el comportamiento del hardware avanzado. Conceptualmente, esta componente es una máquina de estado que podría ser directamente modelada en el hardware.
- Componente de alto nivel: Realizan el control, enrutamientos y toda la transferencia de datos. También realizan cálculos sobre los datos o su agregación.

Las dos fuentes de concurrencia en TinyOS son las tareas y los eventos. Las componentes entregan tareas al planificador con retorno inmediato, se aplaza el cómputo hasta que el planificador ejecute la tarea. Las componentes pueden realizar tareas siempre y cuando los requerimientos de tiempo no sean críticos. Las tareas se ejecutan en su totalidad, y no tiene prioridad sobre otras tareas o eventos. Así también los eventos se ejecutan hasta completarse, pero estos sí pueden interrumpir otros

eventos o tareas, con el objetivo de cumplir de la mejor forma los requerimientos de tiempo real.
Todas las operaciones de larga duración deben ser divididas en dos estados: la solicitud de la operación y la ejecución de ésta. Específicamente si un comando solicita la ejecución de una operación, éste debe retornar inmediatamente mientras que la ejecución queda en mano del planificador, el cual deberá señalizar a través de un evento, el éxito de la operación.

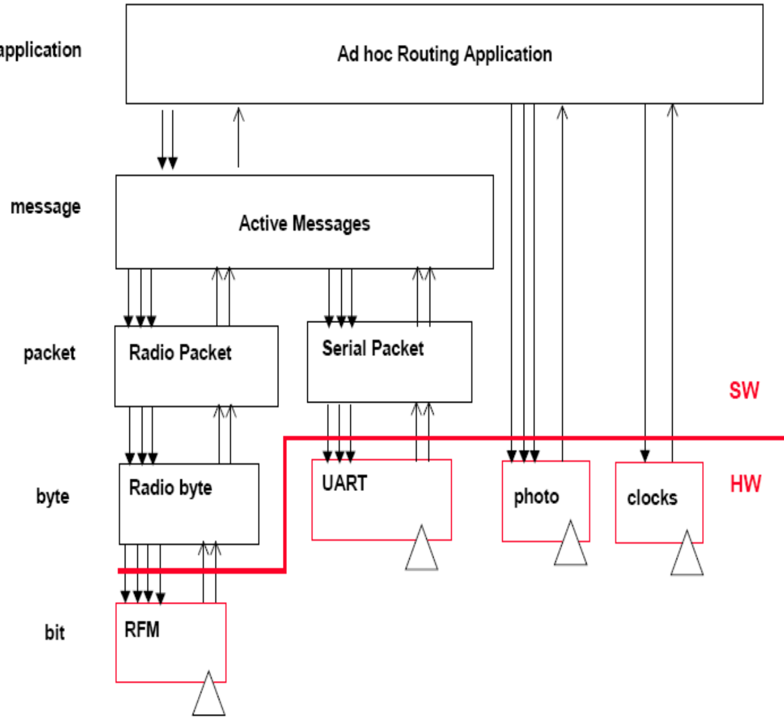


Figura 3.8: Ejemplo de modelo de componentes de una aplicación a varios niveles. *Cada componente se representa como una caja, están interconectadas entre ellas.*

En TinyOs existe una arquitectura software disponible para el microcontrolador TI MSP430, sin embargo, no es el caso del modulo de radio usado en el sensor (Nordic nRF2401). Por lo que se creó una capa HPL (Hardware Presentation Layer) que fue directamente acoplada con una conexión basada en un protocolo MAC.

Pero ésta no permite cambiar de un protocolo MAC a otro, siendo necesaria una arquitectura más modular, que permita conseguir la portabilidad de los protocolos MAC entre las diferentes plataformas HW y de HW a SW, así como la comparación entre protocolos MAC desarrollados por grupos de trabajo diferentes para manejar la optimización de consumo en la capa MAC.

La arquitectura que se requería, más modular, se representa en la *Fig. 3.9*. A continuación se detallarán sus características de sus componentes brevemente:

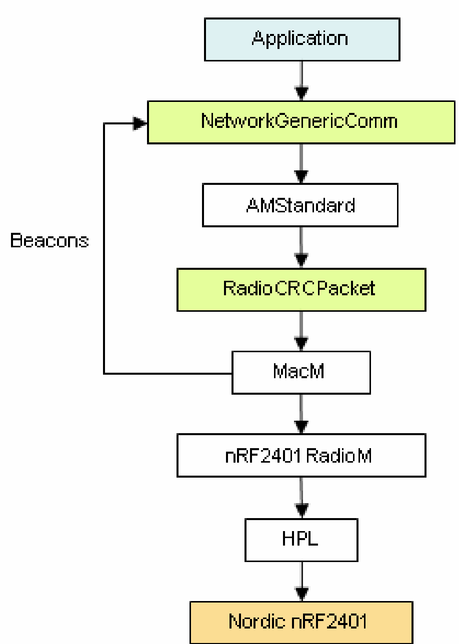


Figura 3.9: Estructura del nRF2401 para TinyOS 1.0. Los bloques en verdes son configuraciones usadas por el S.O para conectar los diferentes componentes, los bloques blancos.

El ***NetworkGeneriComm*** define los componentes usados por USART y la radio de comunicación, y los enlaza con el componente ***AMStandard***.

El componente ***AMStandard*** encapsula los paquetes de la radio a un formato AM (Active Message). La comunicación de la radio en TinyOS sigue el modelo de Mensajes Activos (AM) en el que cada paquete de la red especifica el manejador que será invocado en los nodos.

RadioCRCPacket es el puente entre la capa de red y la capa MAC.

El componente ***MacM*** define la capa MAC y comunica con la capa inferior mediante 4 interfaces: *StdControl, BareSendMsg, ReceiveMsg and MacStdControl*. Las primeras tres son las interfaces por defecto de TinyOS mientras que la tercera ha sido creada para controlar los modos de energía de la radio desde la capa MAC. La capa MAC intercambia paquetes AM con la capa de abstracción hardware a través de *BareSendMsg* y *ReceiveMsg*. Además, el componente *MacM* esta conectado también al *AMStandard* debido a los mensajes de control, que también son enviados en formato AM.

El componente ***nRF2401RadioM*** define la capa de abstracción hardware (HAL) de la radio. Provee a la capa MAC con las funciones de inicialización, envío y recepción. La HAL recibe paquetes AM de la capa MAC, los divide en bytes y los envía a la capa de presentación hardware (HPL). Se hace lo mismo para los paquetes recibidos. Todas las funciones de control de los estados de energía de la radio no se desarrollan en esta capa, de ahí que las llamadas de la capa MAC se pasen al nivel HPL.

El ***HPL*** enlaza el componente hardware con el HAL y ejecuta todas las operaciones necesarias para controlar el dispositivo de radio a nivel de registro.

En la siguiente sección se detallan los protocolos MAC característicos de las redes de sensores, con algunos ejemplos, para el ahorro del consumo, así como el protocolo MAC usado para este proyecto.

3.2.2 TOSSIM

TOSSIM [23] es el simulador original para TinyOs. Se utiliza para calcular el comportamiento de los protocolos y asignaciones de ruta MAC. El simulador puede ser usado para redes prototipadas con muchos nodos (es posible prototipar una red con más nodos que el numero disponible de nodos físicos).
Gracias a esto, el simulador puede calcular la escalabilidad de aplicaciones de redes de sensores inalámbricos.

Otra ventaja de TOSSIM es que puede ejecutar aplicaciones de TinyOs sin necesidad de cambiarlas o modificarlas. También simula el código de forma nativa. Estas características le aventajan sobre los demás emuladores existentes para redes de sensores inalámbricos.

Inicialmente TOSSIM fue creado para la plataforma Mica que tiene un dispositivo de radio transmisor/receptor de 40kbit RFM. En el manual de TOSSIM [23] se presentan los Modelos de Radio, a continuación se resumen esas ideas:

Hay 2 modelos de Radio:

- *Simple*, donde no hay pérdida de señal.
- Modo de Perdidas (*Lossy*), en el que describimos la topología usando un grafo, con aristas entre los nodos x e y si y puede escuchar a x. Las aristas tienen probabilidades de perdida de bits. Para generar tasas de perdida, se usa una herramienta Java *LossyBuilder* que las genera basándose en una topología física.

El modelo de radio de la plataforma Mica2 es implementado en el fichero: tinyos1.x\tos\platform\pc\CC1000Radio\rfm_model.c. Ahí se lee el fichero (por defecto, llamado "*lossy.nss*") con las tasas de perdida generadas por *LossyBuilder*.

El modelo de radio usada para el modelo de nodo en el que se centra este proyecto se encuentra en la carpeta: tinyos1.x\tos\platform\pc\nRF2401RadioShockBurst

3.2.3. PowerTOSSIM

PowerTOSSIM [24] es una extensión de TOSSIM que añade una estimación del consumo de energía para la plataforma Mica 2. En el Apéndice II, se detallarán más características y evolución de esta extensión, ya que para el sensor usado en esta practica, se han modificado algunos ficheros, debido a que el modelo de radio no era el mismo que el de Mica2.

4 Protocolos de control de acceso al medio

4.1 Introducción

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, etc. Además, los nodos pueden fallar en cualquier momento, 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.

Estos protocolos afectan directamente a la disipación de la energía, ya que son la capa más próxima al nivel físico. 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. Existen varias versiones: Basadas en contención sin coordinación., Basadas en planificación, con un nodo central o punto de acceso, encargado de sincronizar al resto, etc.

Los diseños de los protocolos de control de acceso al medio (MAC) varían mucho según el objetivo de la aplicación. Algunos son centralizados, con una estación central como líder del grupo haciendo el control de acceso, otros son distribuidos. Unos usan un único canal, otros varios. Algunos usan diferentes versiones de acceso aleatorio y otros usan reserva de canal y planificación.

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.

Varios protocolos MAC han sido desarrollados para enfrentarse al agotamiento de las baterías, consumo de energía, como S-MAC o T-MAC. A continuación, se describen brevemente:

S-MAC

Es el primer protocolo de la capa MAC desarrollado para redes de sensores, y ha tenido muy en cuenta las limitaciones de energía de estos.
La principal idea es ahorrar energía encendiendo y apagando periódicamente las radios de los nodos, en lugar de tener las radios siempre escuchando posibles transmisiones.

Figura 4.1: División del tiempo en periodos de escucha y sleep. Los nodos sólo pueden recibir mensajes en la fase de escucha. Durmiendo el resto del tiempo, los sensores ahorrarán energía.

Divide la estructura de envío, recepción de mensaje en períodos de escucha y *sleep* (durante el periodo de *sleep* el sensor usa el mínimo de energía). El período de escucha esta dividido en un período de sincronización y otro de transferencia de datos.

Se requiere sincronización, que permite a los nodos anunciar periódicamente su planificación de tiempos (los nodos vecinos se intercambiarán mensajes de sincronización periódicamente (SYNC)), corrigiendo así los desplazamientos temporales propios de un sistema impreciso. Además, permite a la red sincronizar los períodos de *sleep* de los nodos, haciendo así que todos los nodos activen al mismo tiempo los períodos de escucha y *sleep*.

Este protocolo será eficiente energéticamente para baja tasa de envío y alta latencia, ya que los receptores duermen durante bastante tiempo.

T-MAC (Timeout MAC)

Es similar al S-MAC, con alguna mejora, tiene un período de escucha adaptativo basado en el tráfico de la red (tiene ciclos de trabajo de diferente longitud). Con T-MAC, los nodos pueden ir directamente a la fase de *sleep* tan pronto como el tráfico de la red se termine.

La idea es mandar mensajes en ráfagas al principio de este tiempo de trabajo y volver a la fase de *sleep*, tan pronto como no haya más mensajes para ser enviados o recibidos. Los nodos esperarán por un periodo muy corto de tiempo.

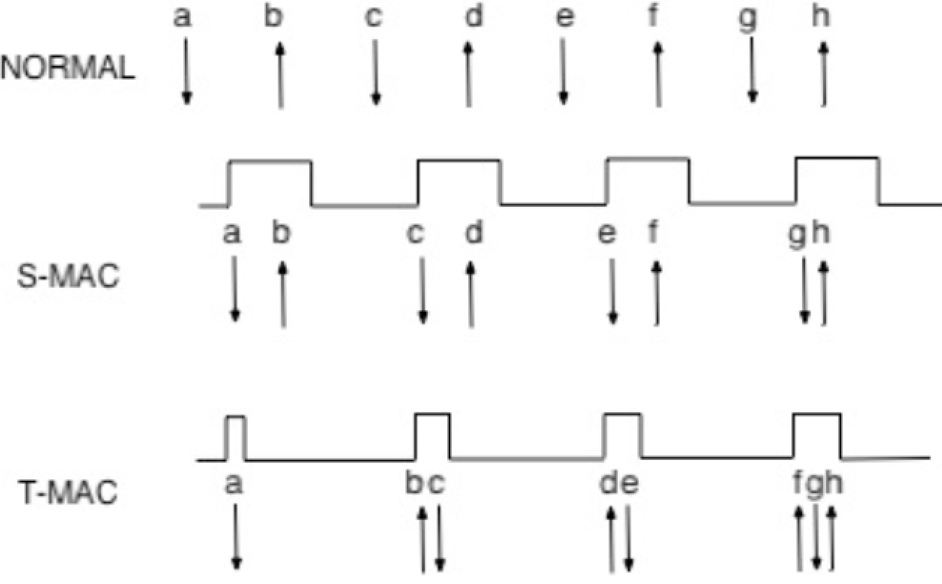


Figura 4.2: Comparativa entre T-MAC y S-MAC

En resumen, un protocolo MAC que consuma lo mínimo para una red de sensores inalámbrica es importante que cumpla lo siguiente:

1- Las colisiones deben ser evitadas siempre que sea posible, ya que la retransmisión produce un innecesario consumo de energía y además posibles retrasos asociados. Por otro lado el evitar las colisiones puede producir una sobrecarga mayor en la red, lo que consumiría mayor energía. Hay que encontrar un punto intermedio, que mejore el consumo.

2- La transmisión de sobrecarga en el protocolo debe ser reducida tanto como sea posible, lo que incluye los paquetes dedicados al control de la red y los bits de cabecera de los paquetes de datos.

3- En los sistemas inalámbricos típicos, el receptor ha de ser encendido siempre, resultando un consumo de energía significativo. Esto es más importante en una radio de corto alcance que en un sistema inalámbrico típico, como una red de computadores o telefonía móvil, ya que un gran porcentaje de la energía consumida ocurre cuando la radio está encendida. La situación ideal sería cuando la radio sólo se enciende cuando necesita enviar o recibir paquetes.

4.2 Protocolo MAC en el nodo: TDMA.

Se han implementado varios protocolos MAC (Dinámico y Estático) para poder hacer comparaciones entre ellos de consumo de energía.

Se eligió un protocolo TDMA (Acceso Múltiple por división de tiempo) que permite a varios usuarios compartir el mismo canal dividiendo el tiempo en diferentes *slots*. Esto

permite que múltiples estaciones compartan el mismo medio de transmisión (como por ejemplo, un canal de radiofrecuencia)

(a) TDMA Estático

Estación base	SB	R	SB	R	SB	R						
Nodo	RB			RB			SSR		RB		S	
Nodo2	RB			RB				RB				
Nodo3	RB		SSR		RB	S			RB	S		

(b) TDMA Dinámico

Estación base	SB	ES	SB	ES	SB	ES	R	SB	ES	R	R
Nodo 1	RB		RB		RB			RB			
Nodo 2	RB		RB	SSR	RB		S	RB		S	
Nodo 3	RB		RB		RB	SSR		RB			S

Figura 4.3. La estación base (BS) manda mensajes de control regularmente para señalizar el comienzo del ciclo TDMA a los nodos.

Cada nodo tiene un *slot* asignado, si un nodo tiene un dato que mandar, éste será enviado en ese *slot*.

Protocolo TDMA Estático

Al comienzo, todos los *slots* están libres y cada nodo debe mandar una petición de slot a la estación base. La estación base responderá alas peticiones de los nodos en el siguiente mensaje de control.

Después, cada nodo enviará una petición de slot cuando se encienda, que se reenviará si tras un tiempo, la estación base no le ha asignado ningún slot a dicho nodo.

Una vez que el slot es asignado, el nodo transmitirá/recibirá los datos a la estación base en ese slot y recibirá las confirmaciones de esas transmisiones/recepciones en los mensajes de control. Si la confirmación no se recibiera tras un determinado tiempo, el nodo reenviaría el paquete.

En este caso, el número de *slots* es un número fijo y conocido. El problema surge porque los nodos tienen un tiempo de vida corto comparado con los PCs normales, por lo que el número de nodos puede tener una clara disminución en el tiempo.

Se puede ver este comportamiento en la *Fig. 4.3.(a)* En el primer ciclo TDMA, el nodo 3 anda una petición de slot, el siguiente mensaje de control le informará de que el primer slot se le ha asignado. En el segundo ciclo TDMA, el nodo 1 realizará la misma petición y se le asignará el segundo slot, y así sucesivamente hasta completar número de slots definidos inicialmente.

Protocolo TDMA Dinámico.

Se propuso para adaptarse al número de nodos de la red de forma dinámica.

Al comienzo, la estación base manda un mensaje de control y deja un slot vacío después de la transmisión de control. Este slot vacío se usa solo para las peticiones de slot. Cuando un nodo desea pedir un slot, manda su petición en esta slot vacío.

Después, en el siguiente mensaje de control, al nodo se le indica el slot asignado para el.

El slot vacío es siempre el siguiente slot después del mensaje de control, y los *slots* dedicados a transmisión estarán colocados tras él.

En este caso, el tamaño del ciclo de TDMA depende del número de nodos que hayan solicitado un slot para transmisión.

Si dos nodos pidieran un slot a la vez, el paquete enviado en el slot vació recibido en la estación base estaría corrupto y se descartaría. El protocolo evita este problema usando una variable aleatoria que indica el número de ciclos que el nodo tiene que esperar antes de pedir un slot.

Para ver cómo se asignan los *slots* en esta nueva implementación, nos fijaremos en la *Fig. 4.3(b)* En el segundo ciclo TDMA, el nodo 2 manda una petición de slot, el siguiente mensaje de control le informará de que el primer slot se le ha asignado. El tercer ciclo TDMA es mas largo, porque hay un slot dedicado al nodo 2, tras el slot vacío. El resto de nodos realizan las peticiones de la misma manera.

En este proyecto, para el estudio de consumo en el nodo (centrado principalmente en el mejorar el consumo de la radio), las pruebas realizadas se han hecho usando el protocolo MAC dinámico, variando algunos de sus parámetros (como el tiempo de slot) en función del comportamiento del programa.

5. Estudio del Consumo

Como ya se ha comentado, el consumo de energía es uno de los principales problemas del nodo. Debido a su reducido tamaño, no permite grandes baterías, pero se desea nodo pueda funcionar durante largos periodos de tiempo sin tener que cambiarlas con mucha frecuencia.

El componente que más consume en el nodo es la radio *(Fig. 5.1*), por lo que no tendremos en cuenta el consumo del sensor, ya que es constante, ni el del microcontrolador, porque es similar ya que todas las pruebas se realizan para modificaciones del mismo algoritmo. La radio consume entre un 50% y un 90% (dependiendo de la aplicación y el modo de la radio que se use) de la energía consumida por el nodo en total, siendo la principal fuente de consumo. Por lo tanto, para minimizar el consumo total en el nodo, nos vamos a centrar especialmente en reducir el gasto de energía de las transmisiones y recepciones.

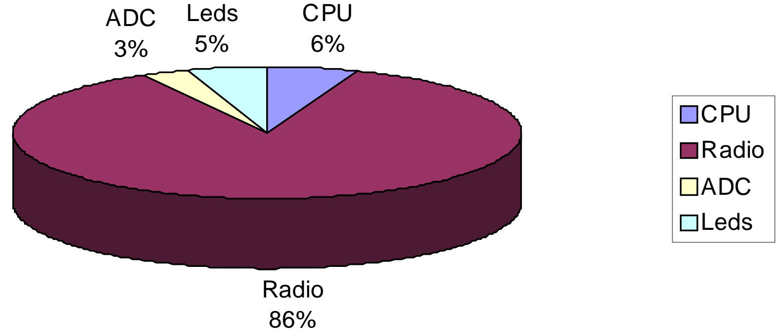


Figura 5.1: Consumo de los componentes del nodo

Se han realizado pruebas de consumo, variando el tiempo de slot del protocolo MAC y modificando la aplicación que se ejecuta sobre los nodos. Estos dos parámetros afectan en gran medida al consumo de energía. El tiempo de slot afecta a la longitud del ciclo TDMA, y consecuentemente al consumo de energía.

Consideramos una red con un número variable de nodos, basada en la implementación dinámica del protocolo MAC. Para este estudio, se simulará una red con 4 nodos y una estación base, tomando el tiempo de slot como parámetro.

Las medidas recogidas se han obtenido a partir de simulaciones, usando una extensión del simulador de TinyOS [22], PowerTOSSIM [24,25], especializado en medir el consumo de energía de cada uno de los nodos de una red, donde se ejecuta una cierta aplicación.

Se ha fijado el tamaño del área de datos del paquete transmitido a 18 bytes por ciclo TDMA. El objetivo es reducir el número de paquetes enviados a la estación base mediante un algoritmo que analiza los datos y envía solo la información necesaria.

Tener un algoritmo para procesar los datos en el nodo, libera a la radio de carga de trabajo, con un pequeño aumento del consumo en el microcontrolador, ahorrando aun así energía. Esto es debido a que el algoritmo se encarga de detectar cualquier anomalía cardiaca en la señal, y solo envía paquetes cuando ha conseguido ha detección completa. Mientras que, hasta ahora, se enviaban todos los datos de la señal ECG a la estación base para que esta los analizara, siendo el número de comunicaciones entre el nodo y estación base mucho mayor.

Al principio de cada ciclo TDMA, la estación base envía un paquete de control a todos los nodos, por lo que todos los nodos están cierto tiempo en escucha debido al protocolo MAC (consumiendo energía). Se medirá el consumo de la radio de los nodos durante 60 segundos, cuando la red esté ya estable.

Algunos de los valores característicos de la simulación son: frecuencia de muestreo de 200 Hz y el nodo tiene un voltaje de 2.8 V.

5.1 Modificaciones para el ahorro de Consumo

Como ya se explicó en apartados anteriores, para minimizar el consumo de la radio del nodo, se han realizado pruebas de consumo usando el simulador PowerTossim [24,25], y las siguientes acciones:

- Incluir un algoritmo de detección de ondas características (apartado 2.2) dentro del nodo y modificar su comportamiento para evitar transmisiones de datos innecesarias. Se han ido consiguiendo estas reducciones en varias versiones que se detallarán más abajo.

- Adaptar el tiempo de slot a los requerimientos de envío de datos de cada versión, en función del comportamiento de esta, para conseguir una reducción importante del consumo en la radio.

Partiendo del enfoque en el que solo se envían los datos sin ser procesados, se pretende minimizar lo máximo posible las transmisiones de los nodos a la estación base. Para esto se han elegido 4 versiones de un algoritmo, pero manteniendo la misma funcionalidad de una a la otra, que se explican a continuación.

Streaming

Este es el enfoque actual, en el que los nodos no usan ningún algoritmo. Únicamente recogen la señal ECG (1 dato cada 5ms) y envían cada uno de estos a la estación base para que la detección y el análisis de las ondas características se realice en ella.

Versión 1

Como ya se ha comentado anteriormente, para reducir el consumo de la radio del nodo, vamos a centrarnos en disminuir las transmisiones de paquetes entre este y la estación base. Para ello se implementó el algoritmo de detección de ondas características de ECG, basado en el artículo de Yan Sun [1] (comentado en el Apartado 2).

Con este algoritmo, ya no es necesario enviar tantos paquetes de datos a la estación base. Y se logra reducir el consumo de energía en el nodo.

La versión inicial del programa implementa el algoritmo de forma que pueda funcionar en el nodo, modificando este para que realizara las detecciones dinámicamente y con pequeñas variaciones que se ajustaran a las limitaciones del nodo (limitado en tamaño de memoria y tiempo de procesamiento), según se explicó en el apartado 2.3.

Se realizó esta modificación, ya que se deseaba que los nodos realizaran las detecciones de forma constante (durante un tiempo indefinido) y en tiempo real. Una vez que se detectan todos los puntos y ondas características de una detección, esta se envía a la estación base (sólo indicando los instantes de tiempo de dichos puntos).

Versión 2

Es posible mejorar aún más el consumo, añadiendo un método en el algoritmo que valide las detecciones.

Una vez que se ha hecho una detección, en lugar de enviarla a la estación base, se puede ver si esta se corresponde con una anomalía cardiaca, mediante las reglas de Normalidad que ya nombramos en el *Apartado 2.4*. En caso de que la detección se corresponda con una señal sin anomalías, no es necesario enviar los datos a la estación base, y sólo si se detectase alguna anomalía en la señal ECG que indique alguna patología cardiaca, se avisa a la estación, mandándole dichos valores de la detección.

Aunque, en el peor de los casos, si todas las detecciones que se realizaran se correspondieran con una anomalía o incumplieran una de las reglas de normalidad, se enviarían todas las detecciones.

Las anomalías aisladas no suelen ser comunes ni significativas, o incluso se pueden deber a un fallo en el análisis de la detección, por tanto se mantendrá un *registro de historia* que guarde el resultado de las ultimas 5 detecciones. Con esto se puede evitar que el nodo envíe avisos o detecciones a la estación base que indiquen alguna anomalía, si esta es puntual. Podemos guardar los datos o resultados de las últimas anomalías detectadas, y en caso de que alguna de ellas sea característica o repetitiva, avisar de ella a la estación base.

Cuando en una detección no cumpla alguna de las reglas de normalidad y si este fallo ya figura en el *registro de* historia dos o más veces y no se trata del mismo fallo que se envió por ultima vez a la estación base, se transmitirá la detección y el resultado de su validación.

Al poder discernir ahora, entre posibles detecciones que muestren un problema cardiaco y detecciones que se correspondan con señales ECG normales, es posible reducir el envío de datos a la estación base aún más, disminuyendo mucho el consumo del nodo, en el mejor de los casos.

Versión 3

Usando el método de validar las detecciones, podemos saber si hay alguna anomalía en el ECG, por lo que no seria necesario mandar los instantes de tiempo de todos los puntos característicos. Sólo es necesario enviar si se ha detectado una anomalía y de que problema cardiaco se trata, ya que los puntos característicos sólo se usarían para ver si se incumple alguna de las reglas de normalidad en el ECG. Si se analiza esto antes de enviar la detección, se puede simplificar la información enviada, y así poder incluir más de una deteccion en un mismo paquete.

En esta versión, por cada detección, solo se mandará el resultado y el instante de tiempo del pico R en un formato de minutos, segundos y milisegundos. Por lo que, la detección solo ocupará 4 datos de 8 bits, siendo posible mandar en un único paquete hasta 4 detecciones (con el instante de su Rpeak y el resultado de estas).

De esta forma, sólo se envía cuando se han detectado 4 latidos, indicando solo el instante de detección de los 4 picos R y el resultado de cada detección.

Versión 4

Podemos reducir más el número de las emisiones, si prescindimos de enviar los instantes de tiempo donde se producen los Rpeaks. Si estos no se envían, se pueden representar más detecciones (solo con el resultado de su validación) en el mismo paquete de datos (hasta un máximo de 18 resultados por paquete).

Los instantes de los picos R son una información muy útil para determinar algunos factores importantes del ECG, como la frecuencia cardiaca. Para compensar la pérdida de información que supone no enviar los instantes de los Rpeaks, se va a enviar también la frecuencia cardiaca calculada en base a esos Rpeaks.

Por lo tanto, se envían en esta versión 17 detecciones en cada paquete, indicando sólo el resultado de cada detección. Los últimos 8 bits del paquete son usados para indicar la frecuencia cardiaca medida a partir de esas 17 detecciones, con la siguiente formula:

Frecuencia Cardiaca = 16 ÷ la diferencia en segundos entre el primer y el ultimo Rpeak.

Con esta ultima versión, además de conseguir el mayor ahorro de energía en la radio, se proporciona el resultado de todas las detecciones y la frecuencia cardiaca del individuo, siendo suficiente, con esta información, para realizar un seguimiento constante.

4.2 Medidas y Pruebas realizadas

En base a las versiones explicadas en el punto anterior, se han hecho modificaciones en el parámetro del protocolo MAC que marca el tiempo de slot.

Vamos a estudiar cuanto es posible aumentar dicho parámetro, dependiendo de cada versión, y después se realizarán simulaciones de dichas versiones fijando el parámetro de tiempo de slot a su valor máximo correspondiente.

Suponiendo en las simulaciones una red con 4 nodos y una estación base, y simularemos durante 60 segundos el comportamiento de esta red una vez este estabilizada. Para ello usaremos la extensión para el estudio de consumo que tiene TOSSIM, el PowerTOSSIM [24,25].

El tiempo máximo de slot se calcula en base a los requerimientos máximos de envío de datos de cada una de las versiones.

Como se usa un protocolo dinámico, cada ciclo TDMA estará dividido en 6 *slots* (el slot para la estación base, el slot vacío, y un slot para cada uno de los cuatro nodos).

Sin usar el algoritmo, enviando todos los datos de la señal (1 dato cada 5 milisegundos), el paquete puede contener como máximo 9 datos, ya que los datos de la señal ECG son de 16 bits. Por lo tanto, se mandan 9 datos cada 45 milisegundos, que dividido entre 6 *slots* de cada ciclo, nos indica que debemos usar un tiempo de slot máximo de 7 milisegundos.

Con la introducción del algoritmo, ese tiempo de slot podrá no ser tan reducido. Aun poniéndonos en el peor de los casos, suponiendo una frecuencia cardiaca máxima de 200 pulsaciones por minuto (ppm) exagerada, en la que habría 200 detecciones en 60 segundos, es decir, 1 detección cada 0,3 segundos o cada 60 datos leídos (en 1 segundo se recogen 200 datos), la detección se mandaría cada 300 milisegundos, que dividido entre 6 *slots* de cada ciclo, nos indica que debemos usar un tiempo de slot máximo de 50 milisegundos.

Por lo que, para la *Versión 1*, en la que se envía todas las detecciones, el tiempo de slot máximo es 50 milisegundos.

Sin embargo, en la *Versión 2*, gracias a que se añade una restricción por la que solo se envían las cardiopatías que se repiten en el tiempo, o son muy comunes, en el peor de los casos, se envía una de cada 3 veces, por lo que se puede ampliar el tiempo de slot a 150 milisegundos.

Las *Versiones 3 y 4*, mejoran aun más el tiempo máximo de slot, gracias a reducir la cantidad de información de cada detección que se envía. Siendo su tiempo máximo de slot de 200 y 850 milisegundos respectivamente (4 y 17 detecciones por paquete respectivamente).

En la t*abla 5.1* se resumen las características de las versiones de este estudio, y el tiempo máximo de slot que permiten sus comportamientos.

	Tiempo max slot (ms)	Descripción
Streaming	7	Se envía cada dato de la señal cuando la recoge el sensor. Sin detecciones.
Versión 1	50	Se envía siempre cuando se ha hecho una detección.
Versión 2	150	Se envía cuando se ha hecho una detección errónea que se ha producido anteriormente varias veces, (en el código es mas de 2 veces), y que no sea el mismo fallo enviado la ultima vez. Registro de Historia de resultados.
Versión 3	200	Se envía cuando se han detectado 4 latidos (complejos qrs) indicando solo el instante de detección de los 4 Rpeaks y el resultado de cada detección.
Versión 4	850	Se envía cuando se han detectado 17 latidos (complejos qrs) indicando solo el resultado de cada detección y la frecuencia cardiaca aproximada.

Tabla. 5.1: Comparativa entre Versiones implementadas para el estudio.

Medidas de Consumo variando el tiempo de slot

Primero, se van a mostrar los resultados obtenidos de la simulación de las versiones, con diferentes valores de slot, para demostrar la gran diferencia que representa para el consumo el tiempo de slot.

Nos centraremos en los valores de consumo de la radio del nodo, ya que es ahí donde se registra el mayor consumo del nodo, los valores de consumo del resto de componentes del nodo son mucho más pequeños.

En la *tabla 5.2* se demuestra la gran reducción del consumo conseguida con las modificaciones hechas sobre el algoritmo.

	Tiempo Slot Máximo (ms)	*Consumo Radio (mJ)*
Streaming	7	421.78
Versión 1	50	60.89
Versión 2	150	19.78
Versión 3	200	15.65
Versión 4	850	3.74

Tabla 5.2. Tabla comparativa de medidas de consumo entre las diferentes versiones. Las medidas están expresadas en milijulios (mj).

Medidas de Consumo variando el número de nodos

Aunque las medidas anteriores se han simulado para una red de 4 nodos, podemos ver como influye el número de nodos en el consumo. Ya que la división del ciclo TDMA

del protocolo dinámico MAC usado en slots, depende del número de nodos de la red, siendo dividido éste en 2+número de nodos *slots*.

Nos vamos a basar en la *Versión 1*, para ver como influye el número se nodos en la red:

Nº nodos	*Consumo Radio(mJ)*
1	238.11
4	120.01
10	118.14

Tabla 5.3. Medidas de consumo para una versión del algoritmo que manda todas las detecciones, en una red de 1, 4 y 10 nodos y una estación base. Las medidas están expresadas en milijulios (mJ).

Usando un nodo y una estación base, podemos permitir un tiempo de slot máximo de 100, ya que en un ciclo TDMA tendremos 3 slots (el de la estación base, el vacío y el del nodo).

Para una red de cuatro nodos y una estación base, podemos tener un ciclo máximo de 50 milisegundos, como ya explicamos al realizar las pruebas de consumo para las versiones del algoritmo.

Sin embargo, en caso de que se use una red con 10 nodos y una estación base, tenemos que dividir el ciclo TDMA entre 12 slots, que para la versión 1, daría un tiempo máximo de slot de 25 milisegundos. Como este es el caso más restrictivo, se fija un tiempo de slot de 25ms para las pruebas de consumo de la versión 1 del algoritmo en una red de 1, 4 y 10 nodos, ya que este tiempo es válido para los tres casos.

Cuanto mayor es el número de nodos en la red, menor es el consumo que se registra en la radio, fijando el tiempo de slot al tiempo máximo permitido para la red con el mayor número de nodos.

Con los resultados obtenidos podemos ver que se ahorra gran cantidad de energía si se limita la comunicación entre nodo y estación base. Se puede prolongar la vida del nodo en funcionamiento sin cambiar las baterías mucho más tiempo en comparación con el funcionamiento de este sin al algoritmo.

En la Tabla 5.2 se aprecia una reducción del consumo de energía en la radio del nodo de más de un 85,56%, dependiendo de la versión del algoritmo que usemos.

El comportamiento del algoritmo ha influido en este ahorro de energía, en versiones en las que se mandan varias detecciones juntas, como la *Versión 3* y 4, se consigue un mejor resultado de consumo de energía, en comparación con versiones que envían un paquete cada detección, como la *Versión 1*.

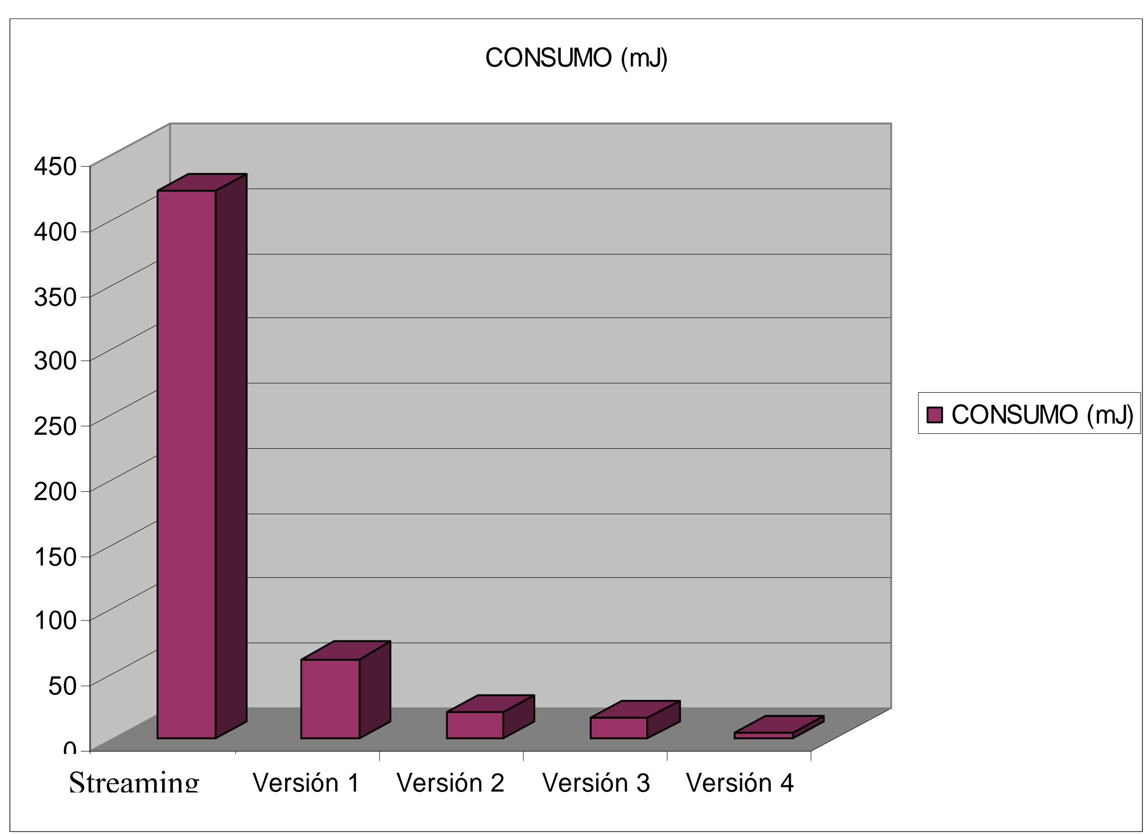


Figura 5.2. Tabla de la relación entre el consumo y las versiones desarrolladas para minimizar el consumo. Medidas toadas en una red de 4 nodos y una estación base con un protocolo MAC dinámico durante 60 segundos. Medidas de consumo en miliJulios.

Cada versión, ha intentado reducir aún más el consumo de energía proponiendo nuevas formas de mantener un seguimiento del estado de la señal ECG de la persona que lleve los nodos. Muchas veces se ha conseguido la reducción de consumo prescindiendo del envío de información que podemos considerar no esencial. De esta forma, es posible un seguimiento mucho más duradero.

También hemos apreciado que la perdida de paquetes del nodo a la estación base se reduce bastante si se usa el protocolo MAC dinámico en vez del estático, esto se debe a la forma de asignar el tiempo dentro del ciclo TDMA a los nodos, como se explicó en las secciones anteriores.

El aumento de nodos en la red, como se ve en la tabla 6, también conlleva cierta variación en el consumo de energía. Pero no se aprecian grandes diferencias en las medidas tomadas, ya que para compararlas se ha tenido que fijar el tiempo de slot al máximo permitido en el caso de la red que más restricciones de este parámetro se exigían.

6. Conclusiones

La minimización del consumo de energía es uno de los principales retos que se plantean en las redes de sensores inalámbricas de área corporal para la monitorización de pacientes, puesto que se desea que el tiempo de vida de los nodos que constituyen la red sea el máximo posible.

En este trabajo se ha diseñado un algoritmo para el análisis de ECG en tiempo real y el diagnóstico automático de posibles patologías cardiacas, que ha sido optimizado para los escasos recursos de procesamiento de una plataforma real.

Se ha probado la precisión del algoritmo, comparándolo con otros que realizan la misma tarea, y se ha demostrado que con el uso del mismo para el preprocesamiento de los datos recogidos por los sensores se consigue reducir el consumo de energía de la radio hasta un 99,11%, con respecto a redes en las cuales toda la información leída por los sensores es enviada directamente a la estación base sin ser procesada previamente.

Esto demuestra la importancia de dotar a los nodos de cierta inteligencia, mediante la inclusión de aplicaciones en los mismos, con lo que la comunicación con la estación base disminuye significativamente y por tanto el consumo de energía se reduce, extendiéndose muy notablemente el tiempo de vida de la red.

APÉNDICE I:

Herramientas Usadas

1. Cygwin

Para la ejecución de todos los programas y plataformas usadas en Windows, se ha usado esta shell que simula plataformas UNIX. Es un entorno desarrollado por Cygnus Solutions para proporcionar un comportamiento similar a los sistemas Unix en Windows.

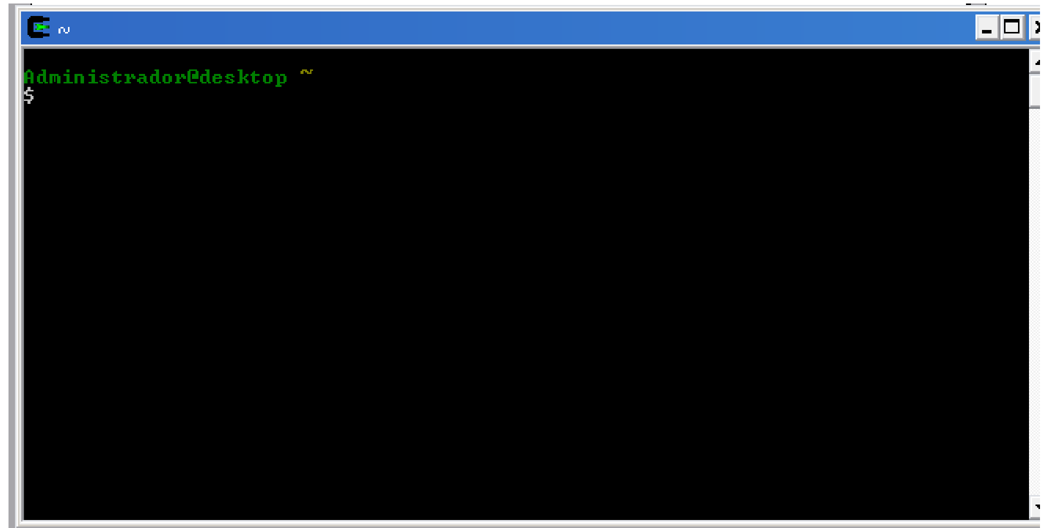


Figura 1: Captura de Cygwin

2. Terminal

Para visualizar los paquetes recibidos en la estación base, y comprobar el correcto funcionamiento del algoritmo en el nodo, así como la recepción y transmisión de paquetes, se ha usado este programa. Los paquetes se muestran en formato hexadecimal, decimal o binario.

Para configurar el programa para recibir correctamente los datos del nodo se fijan las siguientes opciones a estos valores: Baud rate = 57600, Data bits = 8, Parity = none, stop bits =1 y Handshaking = none. Como vemos en la *Fig.24*

Com Port es el puerto al que se conecta la estación base en el ordenador. Para iniciar la conexión y poder ver el contenido de los datos del paquete, sólo es necesario pulsar *Connect*.

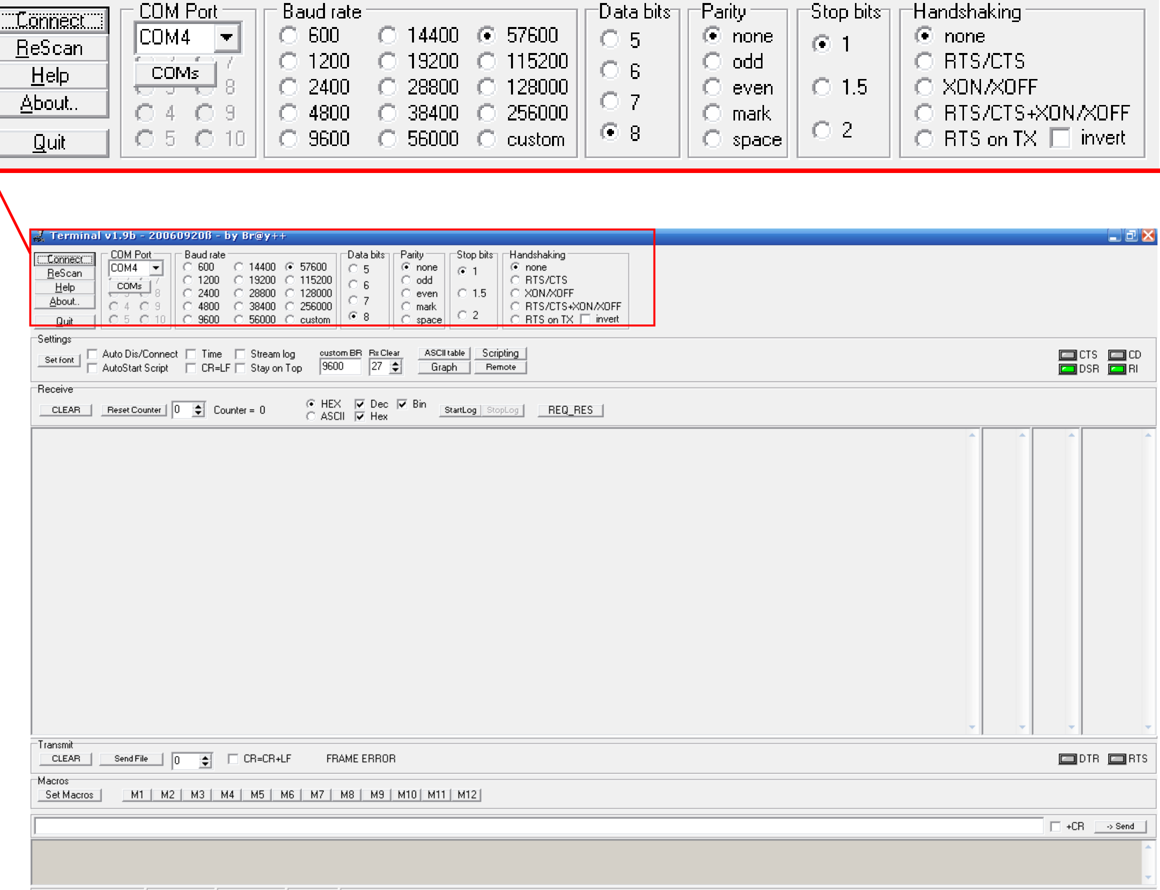


Figura 2: Captura del programa Terminal.

3. *PowerTOSSIM*

PowerTOSSIM [24, 25] es una extensión de TOSSIM que añade una estimación del consumo de energía para la plataforma Mica 2. En el Apéndice II, se detallarán más características y evolución de esta extensión, ya que para el sensor usado en esta practica, se han modificado algunos ficheros, debido a que el modelo de radio no era el mismo que el de Mica2.

Los comandos para la ejecución de las pruebas realizadas en el estudio del consumo de este proyecto son los siguientes, usando Cygwin desde la carpeta donde éste el programa que se desee simular:

Primero, se activa la opción de consumo en el modo debug:

```
export DBG=power
```

Se compila el programa:

```
$ make pc
```

Se simula el programa (se puede indicar el tiempo que se desea simular, el número de nodos, etc.) en este caso, para 5 nodos y 120 segundos. Se guarda el contenido de la simulación en el fichero myapp.trace. Hacer la simulación lleva bastante tiempo:

```
./build/pc/main.exe -t=120 -p 5 > myapp.trace
```

Después, se obtienen los resultados de consumo de la simulación realizada, guardándolos en el fichero consumo.txt:

```
$TOSROOT/tools/scripts/PowerTOSSIM/postprocess.py   -sb=0   --em
$TOSROOT/tools/scripts/PowerTOSSIM/imec_eeg_energy_model.txt
myapp.trace >consumo.txt
```

Para poder hacer varias mediciones o comprobaciones, podemos modificar los parámetros del protocolo MAC dinámico que usa la simulación. Entre los múltiples parámetros del protocolo, en este estudio, hemos modificado el tiempo de slot.

5. IAR Embedded Workbench IDE

Este programa es un Integrated Development Environment (IDE) [26] para la compilación y la depuración de aplicaciones para el microcontrolador MSP430.

Se ha usado en este proyecto para cargar los programas en el nodo y asegurar su correcto funcionamiento. En el apéndice II se detallan las instrucciones para cargarlos y las características que tiene.

Hay dos opciones de ejecución, usando el protocolo dinámico MAC o el estático, para ello se usan una de las dos instrucciones siguientes respectivamente:

```
make ucm_eeg_dyn install.1
```

```
make ucm_eeg_sta install.1
```

Tras ejecutar el programa, se crea una carpeta, dentro de la que contiene los ficheros del programa, llamada build. Esta carpeta a su vez, contendrá otra llamada ucm_eeg_dyn o ucm_eeg_sta, dependiendo del modo del protocolo que hayamos ejecutado. El fichero main.ihex.out-1que hay dentro de ellas es el que usaremos para cargarlo en el modo debbug.

Para este el proyecto, se ha comprobado que el funcionamiento de las comunicaciones con ambos modos del protocolo MAC, obteniendo mejores resultados de con el protocolo dinámico.

APÉNCICE II:

Manuales de Instalación y Uso

1 Instalación de TinyOS y MSP430-GCC en Windows

La instalación debe realizarse en la cuenta de Windows donde se vaya a trabajar posteriormente, ya que si no, no se permite el acceso a los ficheros. Se deben seguir los siguientes pasos por orden. A la hora de guardar los ficheros y elegir las carpetas para instalar el programa es conveniente hacerlo directamente en el directorio raíz para evitar tener rutas excesivamente largas o con espacios que dificultan la instalación.

Archivos requeridos, disponibles en la wiki del proyecto:

- Tinyos-1.1.0-1is.exe
- nesc-1-1-2b-1.cygwin.i386.rpm
- cygwin1.dll (version 1005.18.0.0, July 2005, 1266 KB)
- tinyos-1.1.13May2005cvs-1.cygwin.noarch.rpm
- ucl-dcnds-tinyos-imec.patch
- mspgcc-win32tinyos-20041204-1.cygwin.i386.rpm
- cygintl-3.dll
- cygiconv-2.dll

1. Instalar Tinyos-1.1.0-1is.exe. Este ejecutable incluye Cygwin, Java y TinyOS 1.1.0. Tarda algún tiempo en instalarse.

2. Abrir CygWin. A continuación se instalarán el resto de componentes, escribiendo los siguientes comandos:

   ```
   bash$ cd /cygdrive/c/TinyOS_Holst/
   ```

   Esta ruta corresponde a la carpeta donde tenemos guardados todos los archivos requeridos para la instalación, se debe cambiar la ruta si el directorio raíz no es c.

3. Actualizar la versión de NesC (se requiere antes de que TinyOs pueda ser actualizado):

   ```
   bash$ rpm --force --ignoreos -Uvh nesc-1-1-2b-
   1.cygwin.i386.rpm
   ```

4. Buscar todas librerías cygwin1.dll y reemplazarlos por el que se incluye en esta carpeta (especialmente el instalado en versiones previas de mspgcc/bin). Si no se hace bien, dará el siguiente error: `about_getreent not found in cygwin1.dll.`

5. Reiniciar CygWin, así evitaras posibles fallos.

6. Actualizar TinyOS a la version 1.1.13. Esto también tarda un rato, y un error si el punto 4 no se ha realizado correctamente. Aún así, se producen algunos fallos de compilación en algunos módulos pero esto no es ningún problema para nuestra aplicación.

   ```
   bash$ rpm --force --ignoreos -Uvh tinyos-1.1.13May2005cvs-1.cygwin.noarch.rpm
   ```

7. Aplicar el parche de IMEC:

   ```
   bash$ cd $TOSROOT
   bash$ patch -p0 </cygdrive/c/TinyOS_Holst/Patch_from_UCL/ucl-dcnds-tinyos-imec.patch
   ```

8. Recompilar *motelist*

   ```
   bash$ cd $TOSROOT/tools/src/motelist
   bash$ make clean
   bash$ make install
   ```

9. Recompilar las herramientas de Java. Esto tarda unos minutos.

   ```
   bash$ cd $TOSROOT/tools/java
   bash$ make
   ```

10. Instalar alguna versión reciente de mspgcc.

   ```
   bash$ rpm --force --ignoreos -Uvh mspgcc-win32tinyos-20041204-1.cygwin.i386.rpm
   ```

11. Definir la ruta de la carpeta bin de mspgcc para asegurar su uso correcto. Añadir la siguiente línea al inicio del fichero *c:/tinyos/cygwin/etc/bash.bashrc*.

   ```
   export PATH=/usr/local/mspgcc/bin:$PATH
   ```

   Reiniciar CygWin de nuevo.

12. Copia cygint1-3.dll y cygiconv-2.dll en la carpeta c*:/tinyos/cygwin/usr/bin.* Esto son dll nuevas que requieren las herramientas de mspgcc.

Una vez hecho esto, queda terminada la instalación básica. Pero para completar la instalación para que funcionen las herramientas usadas en este proyecto se necesitan incluir algunos ficheros dentro de CygWin.

Para añadir a TOSSIM el modelo de monitorización de la red: Incluir *NetworkGenericComm* dentro de la carpeta “C:\tinyos\cygwin\opt\tinyos-1.x\tos\lib\”

Para realizar el estudio de consumo con PowerTossim hay que incluir los siguientes ficheros en las carpetas indicadas:

- imec_eeg_energy_model.txt→C:\tinyos\cygwin\opt\tinyos-1.x\tools\scripts\PowerTOSSIM
- node-2.dat→C:\tinyos\cygwin\opt\tinyos-1.x\tools\scripts\PowerTOSSIM
- nRF2401RadioShockBurst → C:\tinyos\cygwin\opt\tinyos-1.x\tos\platform\pc

Instalación la plataforma imec_eeg para compilar con *make install*, según se explica en el apéndice C. Esta es la plataforma de definición de TinyOS para el nodo EEG/ECG de 25 canales. Es necesario incluir:

- ucm_eeg_dyn → C:\tinyos\cygwin\opt\tinyos-1.x\tos\platform
- ucm_eeg_sta → C:\tinyos\cygwin\opt\tinyos-1.x\tos\platform
- ucm_usb_dyn → C:\tinyos\cygwin\opt\tinyos-1.x\tos\platform
- ucm_usb_sta → C:\tinyos\cygwin\opt\tinyos-1.x\tos\platform

Y hay que crear nuevos ficheros que son una modificación de imec_eeg.target y imec_usb.target, todo en el fichero especificado:

- ucm_eeg_dyn.target → C:\tinyos\cygwin\opt\tinyos-1.x\tools\make
- ucm_eeg_sta.target → C:\tinyos\cygwin\opt\tinyos-1.x\tools\make
- ucm_usb_dyn.target → C:\tinyos\cygwin\opt\tinyos-1.x\tools\make
- ucm_usb_sta.target → C:\tinyos\cygwin\opt\tinyos-1.x\tools\make

Dentro de cada uno hay que realizar la siguiente modificación. Este ejemplo es válido para el fichero ucm_eeg_dyn, pero habría que hacer lo equivalente para los otros 3.

```
PLATFORM = ucm_eeg_dyn
MSP_MCU = msp430x149
ucm_eeg_dyn: $(BUILD_DEPS)
```

Por último, modificar el fichero *all.target* dentro del fichero *C:\tinyos\cygwin\opt\tinyos-1.x\tools\make* y añadir la nueva plataforma modificando la siguiente línea:

```
PLATFORMS ?= mica mica2 mica2dot telos telosb micaz pc imec
imec_usb imec_eeg
```

2. *Manual de PowerTOSSIM*

PowerTOSSIM es una extensión de TOSSIM que añade una estimación del consumo de energía para la plataforma Mica 2. Dicha plataforma esta compuesta por:

– Procesador 7.3 MHz ATmega128L
– Memoria de instrucciones de 128KB, 512KB EEPROM
– Memoria de datos de 4KB
– Radio ChipCon CC1000 capaz de transmitir a 38.4 Kbps con un rango de transmisión externo de aproximadamente 300m

Para usar la pila de la radio CC1000 (que incluye el protocolo BMAC) es necesario escribir en el fichero Makefile, de la aplicación que se desea compilar, la siguiente línea:

```
PFLAGS += -I%T/platform/pc/CC1000Radio
```

PowerTOSSIM se incluye a partir de la versión 1.1.9 de TOSSIM. Cualquier versión más actualizada de TinyOS incluirá TOSSIM con su extensión para estudio de consumo PowerTOSSIM.

Es importante mencionar que PowerTOSSIM distribuido con TinyOS solo genera mensajes de log para varios de los eventos importantes de los nodos inalámbricos (el nodo esta a la escucha en ese ciclo, la radio empieza a retransmitir, el sensor esta encendido, etc.)

Fundamentalmente, durante la simulación con PowerTOSSIM, no hay conocimiento del nivel de energía de la batería, ni de cuanta energía se consume en un ciclo de reloj, etc. Por lo tanto PowerTOSSIM no puede generar estrategias online de mantenimiento o cuidado del consumo de energía.

Para deducir el consumo de energía a partir de los mensajes del log generados al compilar y simular, hay que ejecutar la herramienta de postprocesado *postprocess.py* (script de Pitón) sobre las trazas resultantes.

Ejemplo simple del uso de PowerTOSSIM

Primero, compilar una aplicación con la instrucción: `bash$ make pc`

Debemos asegurarnos de que el DBG incluya “POWER”. Si no es necesario ningún otro mensaje en el debug, solo habría que indicar ese modo de debug de la siguiente manera (para el shell apropiado, en nuestro caso Cygwin): `bash$ export DBG=power`

Después, ejecutar main.exe con el flag *–p* y guardar la salida en un fichero (con la extensión “.trace”). Este flag indica el número de nodos de la red. Para ejecutar el main.exe durante 120 segundos para una red con 5 nodos, guardando en el fichero “myapp.trace” la salida, se escribiría esta instrucción:

```
./build/pc/main.exe -t=120 -p 5 > myapp.trace
```

La traza contendrá un mensaje log con este formato:
SIM: Random seed is 812500
0: POWER: Mote 0 ADC ON at 645564
0: POWER: Mote 0 RADIO_STATE ON at 645564
0: POWER: Mote 0 RADIO_STATE ON at 645564
0: POWER: Mote 0 RADIO_STATE ON at 645564
0: POWER: Mote 0 RADIO_STATE TX at 672964
0: POWER: Mote 0 RADIO_STATE RX at 724964

Para obtener el consumo de energía de estos mensajes de log generados, hay que ejecutar el strip de Python "*postprocess.py*" en la traza resultante:

$TOSROOT/tools/scripts/PowerTOSSIM/postprocess.py -sb=0 --em
$TOSROOT/tools/scripts/PowerTOSSIM/mica2_energy_model.txt myapp.trace

Este script reprocesa los datos del log obtenidos de la ejecución de la aplicación en TOSSIM. El parámetro *-sb* especifica si los nodos tienen un sensor adjunto. Y *-em* especifica el modelo de energía usado (en el ejemplo, mica2_energy_model.txt). Por defecto, PowerTOSSIM usa el modelo de energía especificado en el fichero energy_model.txt del directorio actual.

Para ver el resto de opciones y para más detalles ejecutar:
bash$ /opt/tinyos-1.x/tools/scripts/PowerTOSSIM/postprocess.py –help

Donde se especifica su uso y todas sus opciones:
USAGE: postprocess.py [-help] [-debug] [-nosummary][-detail[=basename]] [-maxmotes N][-simple] [-sb={0|1}] --em file trace_file

-help: print this help message
-debug: turn on debugging output
-nosummary: avoid printing the summary to stdout
-detail[=basename]: for each mote, print a list of `time\tcurrent' pairs to the file basename$moteid.dat (default basename='mote')
-em file: use the energy model in file
-sb={0|1}: Whether the motes have a sensor board or not. (default: 0)
-maxmotes: The maximum of number of motes to support. 1000 by default
-simple: Use a simple output format, suitable for machine parking

Por defecto, *postprocessor* imprime el total de energía usada por cada componente de cada nodo, como se puede ver en la siguiente salida (En este ejemplo, para el nodo 0):
Mote 0, cpu total: 143.495863
Mote 0, radio total: 2579.446498
Mote 0, adc total: 93.272311
Mote 0, leds total: 0.000000
Mote 0, sensor total: 0.000000
Mote 0, eeprom total: 0.000000
Mote 0, cpu_cycle total: 0.000000
Mote 0, Total energy: 2816.214672

Para orientar PowerTOSSIM al estudio de los sensores WSN:

– Se cambiaron los valores de consumo de los componentes importantes, para ello se creó un nuevo fichero de modelo de energía (imec_eeg_energy_model.txt)

– Se cambió la manera en que se lleva a cabo la simulación debido a las diferencias de conducta entre los nodos mica o mica 2 y los sensores (solo centrándose en las componentes con mayor consumo de energía: la CPU y la Radio) debido a que:

  La CPU de los sensores es un TI MSP430 en lugar de ser un Atmega128L, con código maquina de distinto tamaño (con un numero diferente de ciclos de reloj ejecutados). Por tanto, para una simulación correcta de la CPU se compilará el fichero C generado a partir del fichero NesC, usando el compilador GCC para el TI MSP430.

  Y para la radio, en vez del modelo CC1000, se usa el transmisor-receptor nRF2401. A continuación se explicaran brevemente ambos modelos.

Para compilar una aplicación con un modelo de radio u otro, es necesario añadir en el makefile:
PFLAGS += -I%T/platform/pc/CC1000Radio para el modelo de radio CC1000 (Mica 2)
PFLAGS += -I%T/platform/pc/nRF2401RadioShockBurst para el modelo nRF2401.

3. Manual de IAR Embedded Workbench IDE

Este programa es un Integrated Development Environment (IDE) [26] para la compilación y la depuración de aplicaciones para el microcontrolador MSP430 [27]. El IAR incluye un compilador de C/C++ con limitador de tamaño, establecido de 4kB a 8Kb dependiendo del dispositivo que se quiere probar, pero que no vamos a usar. Y un depurador o simulador es ilimitado con soporte para código complejo y breakpoints.

La forma de cargar programas en el MSP430 del nodo para este proyecto ha sido a través del modo debbug.

Antes de portar el programa en el nodo, para cargar una aplicación se necesita:

- Tener instalado TinyOS.
- Incluir las carpetas y ficheros con nombre ucm_eeg_dyn y ucm_eeg_sta
- La versión 3.41A del programa IAR Embedded Workbench IDE.
- La carpeta "demo" donde se incluye un conjunto de programas de prueba para cargar una aplicación cualquiera al sensor.
- Tener tanto el sensor como la estación base instalados

Abriendo la consola de Cygwin, compilar el programa que se quiere portar desde la carpeta donde estén ubicados los ficheros .nc y .c. Se distinguen dos formas para compilarlo: estática y dinámica, cuya diferencia está en el uso de un protocolo de comunicación MAC estático o dinámico.

```
bash$ cd $TOSROOT/apps/carpetaPrograma
bash$ make ucm_eeg_sta install.1
bash$ make ucm_eeg_dyn install.1
```

Hecho esto, el siguiente paso es abrir la carpeta demo/test/ecg y sustituir el archivo *main.ihex.out-1* por el que se ha creado en la carpeta build/ucm_eeg_sta o build/ucm_eeg_dyn de nuestra aplicación.

Antes de seguir es importante tener instalados tanto el sensor como la estación base que va a recibir los paquetes. Ambos se instalan usando el "asistente de nuevo hardware encontrado" de Windows, especificando la localización de los drivers.

- En el caso del sensor, los drivers están en una carpeta del propio IAR System, en la siguiente ruta: .\IAR Systems\Embedded Workbench Evaluation 4.0\430\drivers\TIUSBFET\WinXP

- Para la estación base es necesario descargarse los drivers de la página http://www.ftdichip.com/Drivers/VCP.htm para el FT232BM, y especificar esa carpeta en el asistente de instalación.

A continuación se procede a cargar el programa ya compilado en el sensor utilizando para ello el entorno de desarrollo IAR Embedded Workbench IDE.

Al abrir el programa, se muestra una ventana de inicio donde se puede crear un nuevo proyecto o abrir uno existente. En nuestro caso vamos a abrir un Workspace ya existente, dentro de la carpeta demo, llamado *test*, que esta desarrollado exclusivamente para trabajar con el sensor de 25 canales.

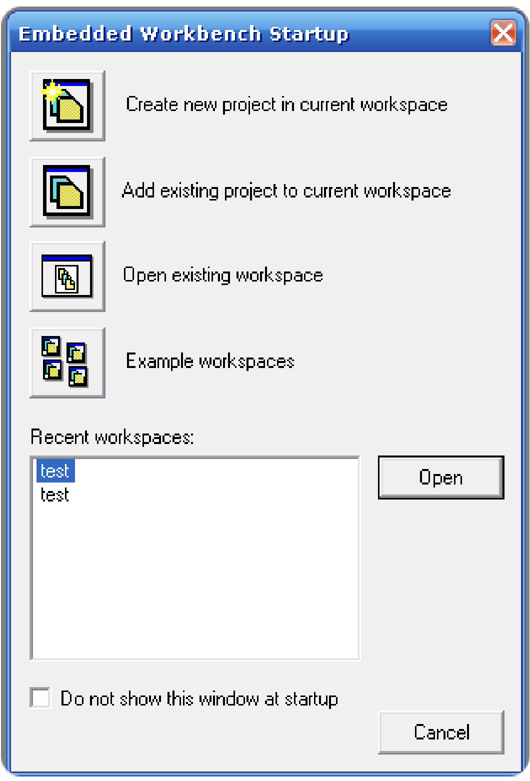


Figura 3: Ventana de inicio.

El WorkSpace tiene que ser previamente guardado en el directorio raíz, en una ruta que no contenga espacios, para evitar posibles errores. Después hay que conectar el nodo a un puerto USB mediante el cable de carga, ya que es imprescindible para depurar las aplicaciones.

Para probar las aplicaciones elegimos la opción Project → Debug.

Se cargará la aplicación en el nodo, lista para ser ejecutada.

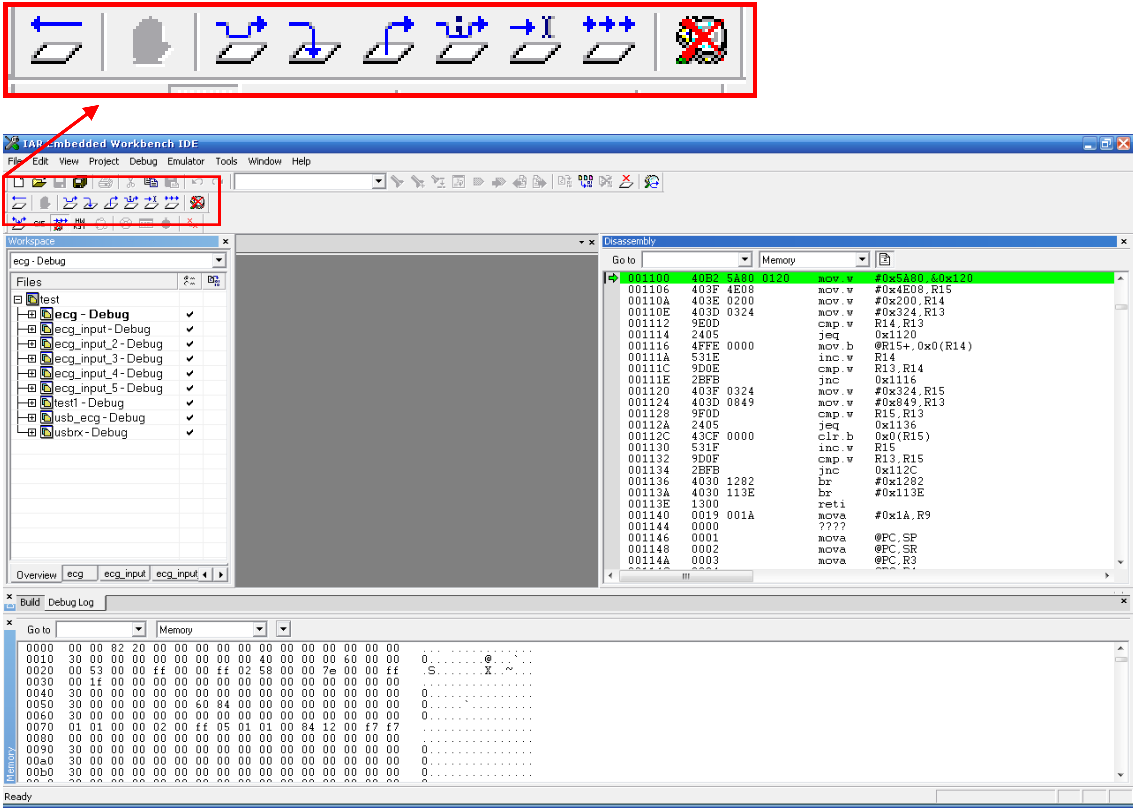
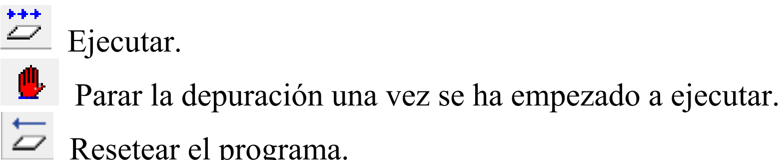


Figura 4: Captura de la ejecución de la aplicación en el nodo.

En el panel inferior se muestra el mapa de memoria del nodo, en el panel derecho el código ensamblador de la aplicación.

En la parte superior se encuentra la barra de herramientas de depuración, con los siguientes botones principales:



Ejecutar.

Parar la depuración una vez se ha empezado a ejecutar.

Resetear el programa.

Este es el proceso que hay que repetir cada vez que se quiera portar una nueva aplicación al nodo.

4. Manual de NesC

1. Introducción a NesC

NesC [6] es un metalenguaje de programación basado en C, orientado a sistemas empotrados en red. Soporta un modelo de programación que integra el manejo de comunicaciones, así como las concurrencias que provocan las tareas y eventos.

Esta especialmente diseñado para soportar el modelo de ejecución de TinyOS (sistema operativo para dispositivos con recursos limitados).

Entre otras ventajas, NesC realiza optimizaciones en la compilación del programa, detectando posibles errores, simplificando el desarrollo de aplicaciones, reduciendo el tamaño del código, y eliminando muchas fuentes potenciales de errores.

1. 1 Características de NesC

Los conceptos básicos que NesC ofrece son:

- Separación entre la construcción y la composición. Las aplicaciones estas formadas por un conjunto de componentes agrupados y relacionados entre sí.
- Hay dos tipos de componentes en NesC: módulos y configuraciones.
    - Los módulos proveen el código de la aplicación, implementando una o más interfaces. Estas interfaces son los únicos puntos de acceso a la componente (implementan especificaciones de una componente).
    - Las configuraciones son usadas para unir (*wire*) las componentes entre sí, conectando las interfaces, que unas componentes proveen, con las interfaces usan otras.

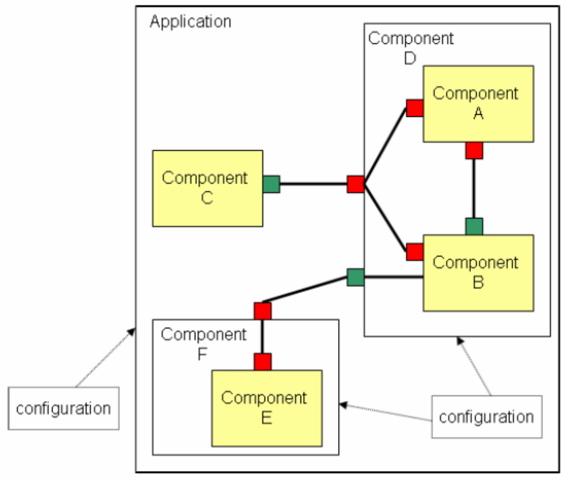


Figura 5: Aplicación como un conjunto de componentes agrupados y relacionados.

- Los interfaces son bidireccionales: las interfaces son los puntos de acceso a las componentes, conteniendo comandos y eventos, los cuales son los que implementan las funciones. El proveedor de una interfaz implementa los comandos, mientras que el que las utiliza implementa eventos. Los comandos son llamadas a componentes de capas inferiores, y los eventos son llamadas a capas superiores, usadas para interactuar con el Hardware.

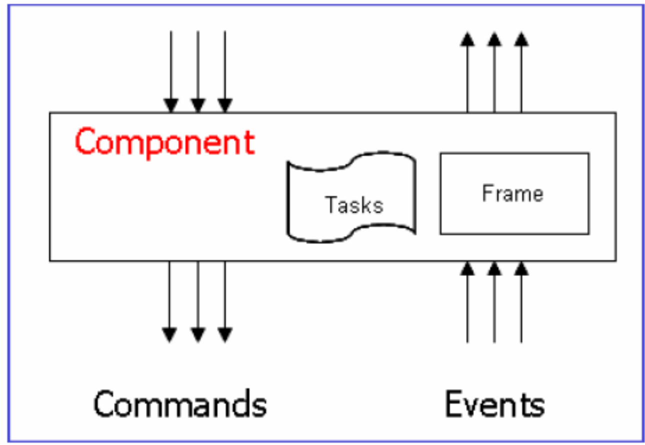


Figura 6: Diagrama de comandos y eventos

- Unión estática de componentes, vía sus interfaces. Esto aumenta la eficiencia en tiempo de ejecución, incrementa la robustez del diseño, y permite un mejor análisis del programa. NesC no permite la programación dinámica.

- NesC presenta herramientas que optimizan la generación de códigos. Un ejemplo de esto es el detector de carreras de datos, en tiempo de compilación.

- El modelo de concurrencia de NesC esta basado en la ejecución completa de tareas y manejadores de interrupciones que pueden interrumpir dichas tareas. El compilador señala las carreras de datos causadas por dichos manejadores.

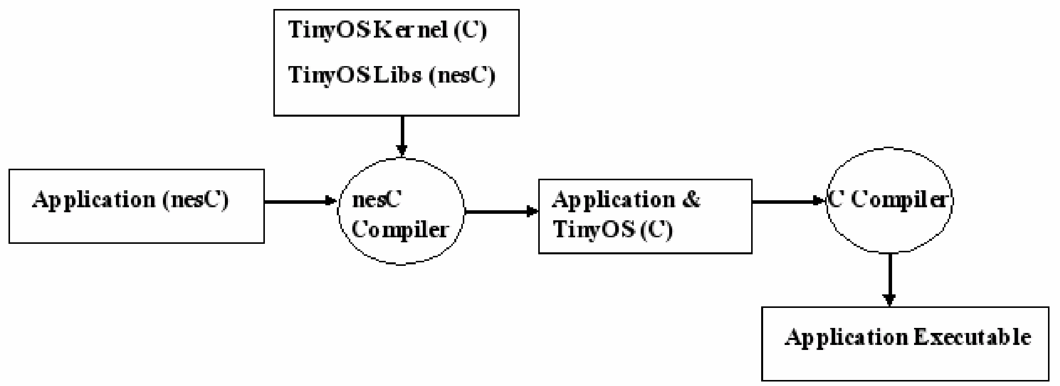


Figura 7: Proceso de compilación de una aplicación TinyOS.

1. 2 Tipos de Datos

NesC tiene definidos los siguientes tipos de datos, además de permitir los tipos de datos de C:

- **uint16_t** es un entero sin signo de 16 BIT
- **uint8_t** es un entero sin signo de 8 BIT
- **bool** es un booleano (TRUE , FALSE)
- **result_t** es un booleano con los valores SUCCES y FAIL

2. Componentes

Los componentes usan interfaces de componentes ya existentes y proporcionan nuevas interfaces para poder ser usadas por otros componentes.
Un componente puede proporcionar interfaces para poder ser utilizadas por otros componentes que hagan de aplicación. Si un componente hace de aplicación deberá proporcionar una interfaz especial StdControl.

2.1 Estructura de un Componente

Físicamente, los componentes se estructuran en 2 ficheros (por convenio) llamados “configuración e implementación” y “módulos”, además de las librerías (.h) que puedan usar.
Estos dos ficheros tienen extensión .nc. Para nombrarlos se sigue el convenio de usar el mismo nombre para ambos, acabando el fichero de módulos con una “M” al final.

Por ejemplo,
“*miaplicacion*.nc” (fichero de configuración e implementación)
“*miaplicacion*M.nc” (fichero de módulos)

Un componente tiene 3 partes lógicas diferenciadas:

- Configuración: Para configurar el componente. Se usa generalmente para crear librerías. Generalmente se deja vacío.
- Implementación o *Wiring*: Se decide que la interfaz que usa una aplicación es la que proporciona un componente.
- Módulos: Código C que define el comportamiento de la aplicación. Que se estructura en 3 partes: `Provides` (interfaces que provee), `Uses` (interfaces que usa) y `Implementation` (comportamiento del modulo)

2.2 Especificación de un Componente

Un componente puede ser una configuración o un modulo. A continuación veremos las dos formas de especificarlo:

2.2.1 Especificación de un componente como un módulo

***Archivo NesC de "miaplicacion*M.nc":**

```
includes [nombre de librerías y .h si es necesario]
module [nombre del modulo] {
      Uses {}
      Provides {}
      }
Implementation {}
```

En `Provides` se especificaran las interfaces que proporciona el componente (el modulo tendrá que tener implementadas las funciones de dicha interfaz)

En `Uses` se especifican las interfaces que va a usar el modulo (en la parte de *Wiring* se establece que modulo proporciona dicha interfaz)

En implementación debemos definir el comportamiento de la aplicación. Esta parte como mínimo debe incluir:

- Las variables globales
- Las funciones de las interfaces que proporciono (`Provides`)
- Los eventos de las interfaces que utilizo (`Uses`)

En un modulo puede haber varios `Uses` y varios `Provides` ya que puede usar o proveer varias interfaces.

Estos `Uses` y `Provides` se pueden especificar en grupo o con varias instrucciones simples:

```
Uses {
Interface X;                    Uses interface X;
Interface Y;}        ⇒          Uses interface Y;
```

Si usamos una interfaz (`Uses`):

- Podemos llamar a sus métodos.
- Tenemos que implementar los eventos que se van a producir por utilizar la interfaz.
- Hay que realizar el *Wiring* en el fichero correspondiente para indicar quien proporciona la interfaz.

2.2.2 Especificación de un componente como una configuración

Un componente se enlaza a otros a través de sus interfaces, estos enlaces se definen en la parte de *Wiring* (`Implementation`) de este archivo de configuración, con la instrucción ->, mas adelante se detallan las variantes para enlazar interfaces.

Suponiendo que se desee enlazar el componente *nombre1* con el componente *nombre2*, cuyos interfaces respectivos son *interfaz1* e *interfaz2*, el *wiring* se haría de la siguiente forma:

***Archivo NesC de "miaplicacion*.nc":**

```
includes [nombre de librerías y .h si es necesario]
configuration [nombre de la configuración] {}
Implementation{
        Components nombre1, nombre2;
        nombre1 . interfaz1 -> nombre2 . interfaz2;
}
```

En la *sección 5* de este manual se explica más detalladamente el wiring y su semántica.

2.3 Resumen de la estructura

Los ficheros de la aplicación que se ha usado de ejemplo en este apartado quedan reflejados en la *Fig. 2.1*, en la que también se ha incluido un tipo enumerado "paquete".

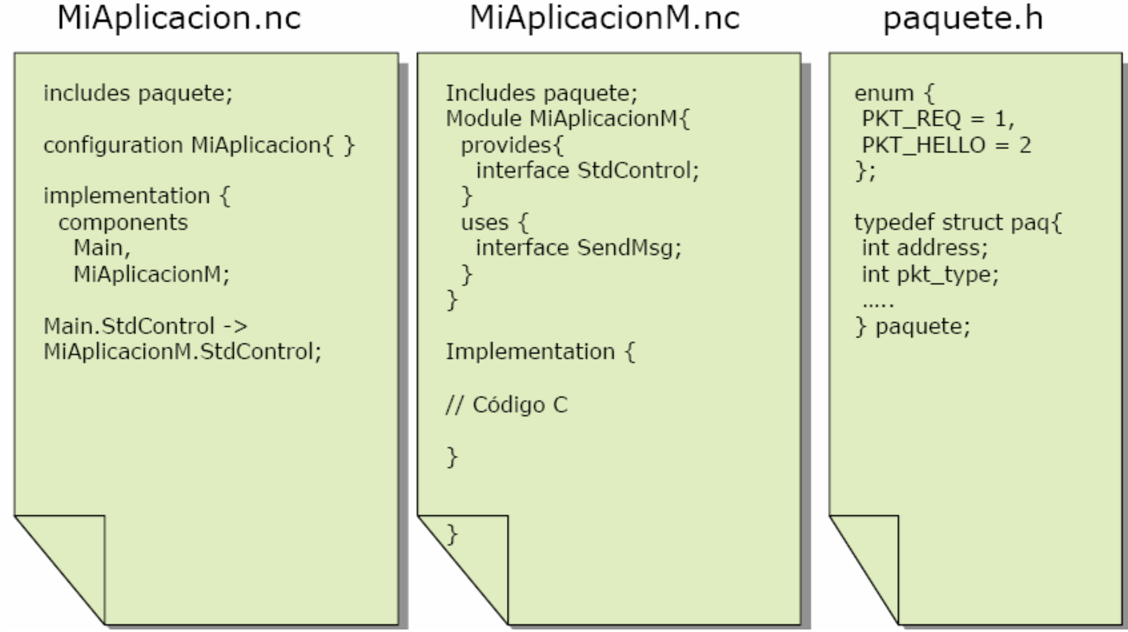


Figura 8: Resumen de la estructura de la aplicación de ejemplo "MiAplicacion"

2.4 Componentes proporcionados por TinyOs

A los componentes que proporciona TinyOs se les llama Primitivos, el otro tipo de componentes son los Compuestos. Los componentes compuestos los proporciona una librería o una aplicación.

Los componentes usados para interconexionado (*InterNetworking*) son los componentes primitivos para redes. Estos se basan en un paquete *TOSMsg* con este formato:

- Dirección de destino (No lleva la dirección de origen)
- Data. (Datos del mensaje)
- CRC
- Longitud
- TOSMsgPrt (puntero a un *TOSMsg*)

Entre este tipo de componentes, se encuentran los componentes *GenericComm* y *GenericCommPromiscuous*. Son usados para enviar y recibir paquetes por radio o UART y proporcionan las interfaces *SendMsg* y *RecibeMsg* (para envío y recepción).
Se comportan como un switch (si reciben un paquete con dirección UART lo envían por el puerto serie y si reciben otra dirección lo envían por radio).

Estas interfaces se explican con más detalle en el *apartado 3*.

Otro componente de gran utilidad es el componente *TimerC*, que ofrece funciones de temporización mediante la interfaz *Timer* parametrizada (para poder tener varios *Timer*, 10 como máximo).

Al utilizar la interfaz *Timer* es necesario implementar el evento `fired()` que se llamara cada x tiempo.

Para arrancar un *Timer* se llama a *start()* con los siguientes parámetros: `Start(tipo,xtiempo);`
Donde `tipo` puede ser: TIME_REPEAT (si se llama al evento *fired()* cada xtiempo) o TIMER_ONE_SHOT (si solo se llama una vez en un xtiempo) y `xtiempo` (en milisegundos) es el tiempo o intervalo en el que se desea ejecutar el evento *fired()*.

Para parar el *Timer* ya arrancado solo hay que llamar a *stop()*.

3. Interfaces

Los interfaces en NesC son bidireccionales, especificando un canal de interacción multifunción entre dos componentes, el que la provee y el que la usa.
La interfaz especifica un grupo de declaraciones de funciones llamadas comandos, para ser implementadas por el proveedor y otro grupo llamadas eventos, que serán implementadas por el componente que use la interfaz.

3.1 Especificación de Interfaces

Los interfaces se especifican por tipos de interfaces. A continuación se declara una interfaz con el identificador "*X*". Este identificador es de ámbito global:

Archivo NesC:

```
interface X {
    command tipo_devuelto nombreComando(lista de parámetros);
    event tipo_devuelto nombreEvento(lista de parámetros);
};
```

No se puede usar el mismo nombre para definir dos interfaces así como tampoco se permite que un componente y un interfaz usen el mismo nombre o identificador.
Dentro de la especificación del interfaz se deben definir las declaraciones de las funciones usando `command` o `event` para los comandos o los eventos respectivamente. En caso contrario ocurrirá un error en tiempo de compilación.
También es posible usar `async` delante de `command` o `event`, de forma opcional, para indicar que el comando o el evento pueden ser ejecutados en el manejador de interrupciones.

Un ejemplo de interfaz simple:

```
interface SendMsg {
 command result_t send(uint16_t addr, uint8_t length, TOS_MsgPtr msg);
 event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

Los componentes proveedores del tipo de interfaz *SendMsg* deberán implementar el comando *send*, mientras que los usuarios de este interfaz implementarán en evento *sendDone*.

3.2 Instancias de Interfaces

A continuación se declara una instancia de una interfaz con el identificador "*X*":

```
interface X [lista de parámetros opcional];
```

Otra opción para declarar una instancia de interfaz seria:

```
interface X as X;
```

También es posible un interfaz especificando explícitamente su nombre.

```
interface X as Y;
```

Siendo *Y* el nombre de la instancia.

Si los parámetros del interfaz se omiten, esta última instrucción declararía una simple instancia de interfaz, correspondiendo una sola interfaz a este componente.

Si los parámetros del interfaz están presentes (como por ejemplo, `interface SendMsg S[uint8_t id]`) será una declaración de una instancia de interfaz parametrizada, es decir, a este componente le corresponderán varios instancias de ese interfaz, una por cada valor que pueda tomar *id* (en este caso, al ser *id* de 8 bits, en el ejemplo se estarían declarando 256 interfaces del tipo *SendMsg*)
El tipo de parámetros de los interfaces debe ser enteros sin signo.
Los comandos o eventos pueden ser incluidos directamente como elementos especificados añadiendo una declaración de función estándar de C con `command` o `event`.
Al igual que en las instancias de interfaces, los comandos o eventos serán simples si no se especifican parámetros de interfaz, y si se especifican parámetros de interfaz, los comandos o eventos serán parametrizados. Por ejemplo, el siguiente comando queda parametrizado al añadir `[uint8 t id]`:

```
command void send[uint8_t id](int x);
```

3.3 Interfaz StdControl

La interfaz StdControl es una interfaz especial, como se ha mencionado anteriormente, si un componente proporciona esta interfaz, puede hacer de aplicación.

Esta interfaz obliga a tener las funciones Init( ), Start() y Stop()

`Void Init()`: Se ejecutará al arrancar el sistema. Inicializa las variables globales y llama a los métodos *init()* de los componentes que utiliza (solos a los necesarios)

`Void start()`: Se ejecutará después del *init()* y cuando el sistema pase de off a on. Arranca los temporizadores y llama a los métodos *start()* de los componentes que utiliza (a los que sean necesarios)

`Void stop()`: Se ejecutará cuando se apague el sistema o se suspenda. Llama a los métodos *stop()* de los componentes que utiliza.

3.4 Interfaz SendMsg

Si usamos la interfaz *SendMsg* de un componente *GenericComm* podemos usar la función: `Send(addres , long_datos , TOSMsg )` que envía el paquete *TOSMsg* (que debe ser una variable global) a la dirección indicada *addres*. Pero se debe implementar el evento `SendDone()`.

Del paquete *TOSMsg* solo se rellena el campo *Data* (con una estructura del tipo de mensaje que se haya elegido)

Existen constantes para definir la variable *addres* como dirección especial:
TOS_BCAST_ADDR indica la dirección de *Broadcast* de red.
TOS_LOCAL_ADDRESS indica la dirección de localhost.
TOS_UART_ADDR indica la dirección del puerto COM.

3.5 Interfaz ReceiveMsg

Si usamos la interfaz *ReceiveMsg* de un componente no podemos usar ninguna función, pero si hay que implementar el evento de recibir un mensaje.

Se recibe un mensaje en el caso de que la dirección destino del mensaje coincida con nuestro TOS_LOCAL_ADDRESS o sea un mensaje de difusión, TOS_BCAST_ADDR

En el caso de ser el Componente *GenericCommPromicouos* siempre recibimos todos los mensajes, sea cual sea la dirección destino.

4. Implementación de la especificación del Modulo.

Cuando especificamos un componente como un modulo, se debe incluir en la parte de `Implementation{}`de este, todos los comandos o eventos provistos por ese modulo, así como también los comandos de los interfaces que provea y todos los eventos de los interfaces que use.

Un modulo puede llamar a cualquiera de sus comandos y señalar cualquiera de sus eventos.

Para implementar estos comandos y eventos se usan extensiones de código C. La implementación de un comando o evento simple tiene la sintaxis de una definición de función en C. Además, la palabra reservada `async` debe ser incluida, si fue también incluida en la declaración del modulo.

Si el comando o evento pertenece al interfaz se indicará el nombre del comando o el evento como *NombreInterfaz* `.` *NombreComando/Evento*. También se puede hacer una declaración por defecto poniendo `default` delante de la palabra `command` o `event`.

Ejemplos de implementación del comando *send()*, en un modulo que provee un interfaz *Send* del tipo *SendMsg*:

```
command result_t Send.send(uint16_t address, uint8_t length, TOS_MsgPtr msg) {
...
return SUCCESS;
}
```

Si la interfaz *Send* estuviese parametrizada, siendo *Send*[uint8_t id] del tipo SendMsg:

```
command result_t Send.send[uint8_t id](uint16_t address, uint8_t length,
TOS_MsgPtr msg) {
...
return SUCCESS;
}
```

Se producirán errores de compilación si hay algún comando o evento provisto sin implementación.

4.1 Llamadas a comandos y señalización eventos

Un comando *X* puede llamarse con la instrucción: `call X(. . .);`

Un comando parametrizado *Y*, con *n* parámetros de interfaz [e1, e2,.. en] es llamado así:

`call Y[e1, e2,.. en](. . .);`

Un evento *X* puede señalarse con la instrucción: `signal X(. . .);`
Un evento parametrizado *Y*, con *n* parámetros de interfaz [e1, e2,.. en] es llamado así:

`signal Y[e1, e2,.. en](. . .);`

Es muy importante que cada parámetro sea asignado a una variable o valor de su mismo tipo. Por ejemplo, En un modulo que usa el interfaz *Send*[uint8_t id] del tipo SendMsg:

```
int x = ...;
call Send.send[x + 1](1, sizeof(Message), &msg1);
```

La ejecución de comandos y eventos es inmediata. Las actuales implementaciones de comandos y eventos ejecutadas por las expresiones `call` y `signal` dependen de las declaraciones de las interconexiones (*wiring*) en las configuraciones del programa. Estas declaraciones pueden especificar que 0, 1 o mas implementaciones tienen que ser ejecutadas. Cuando más de una es ejecutada, se dice que el comando o evento del modulo tiene “fan-out”.

Un modulo puede especificar una implementación por defecto para un comando o evento X que use y que sea llamado o señalizado. Sin embargo, se detectara un error en tiempo de compilación si la implementación por defecto se usa en los comandos o eventos que provee.

Las implementaciones por defecto se ejecutan cuando el modulo no esta conectado a ninguna implementación del evento o comando cuado este se señale o se llame. Para definir un comando o evento por defecto se añade la palabra `default` delante de la palabra `command` o `event`. Como se ve en el ejemplo siguiente:

```
default command result_t Send.send(uint16_t address, uint8_t
length,TOS_MsgPtr msg) {

return SUCCESS;

}
```

Por lo tanto, llamar a este comando cuando el interfaz *Send* no estuviera conectado, estaría permitido.

4.2 Tareas

Una tarea es un contexto de ejecución que corre hasta completarse en background, sin interferir en otros eventos del sistema.

Para definir una tarea y su función se usa: `task void myTask() { ... }.`
Es posible declarar antes la tarea sin devolver argumentos: `task void myTask();`
Para llamar a la tarea e iniciarla usaremos: `post myTask();`

Esta última llamada devuelve un `unsigned char` con valor 1 si la tarea fue planificada correctamente para una ejecución independiente, y 0 en cualquier otro caso.

4.3 Sentencias Atómicas

Estas sentencias que llevan delante de ellas la palabra `atomic`, garantizan que la su ejecución se realiza sin ninguna otra computación simultanea.

Suelen usarse para implementar la exclusión mutua, actualizar estructuras de datos concurrentes, etc.

Un ejemplo de sentencias atómicas en una función es este:

```
bool ocupado; // global

void f() {
      bool disponible;
      atomic {
            disponible = ! ocupado;
            ocupado = TRUE;
      }
      if (disponible) hacer_algo;
      atomic ocupado = FALSE;
}
```

NesC prohíbe llamar a comandos o señalar eventos dentro de la sección de sentencias atómicas, ya que esta sección debería ser corta. Las siguientes instrucciones también se prohíben dentro de la sección atómica: `goto`, `return`, `break`, `continue`, `case`, `default`, y `labels`.

5. Wiring

El interconexionado o wiring se usa para conectar los elementos especificados como interfaces, comandos y eventos. En esta sección se definirá la sintaxis y las reglas de compilación para el interconexionado.

5.1 Sintaxis

Las sentencias de conexionado unen dos puntos finales. El identificador de camino de un punto final especifica un elemento, que opcionalmente puede tener parámetros de interfaz.

Sintaxis de tipos de conexiones: a `=` b, a `->` b, a `<-` b (siendo a y b *puntos finales*)
Una conexión puede estar compuesta por una lista de conexiones. Cada conexión tiene lugar entre dos "*puntos finales*". Un punto final puede ser un identificador de camino, con o sin parámetros de interfaz.

5.2 Reglas de compilación para el interconexionado

Hay 3 sentencias de conexión distintas en NesC:

- Endpoint1 `=` Endpoint2 (Equate wires) Cualquier conexión conlleva un elemento de especificación externo, esto equivale a dos los dos elementos especificados equivalentes.

  Siendo S1 el elemento de especificación de Endpoint1 y S2 el de Endpoint2, se dará un error en tiempo de compilación si no se mantienen una de las dos siguientes condiciones:
  - S1 es interno y S2 externo (o viceversa) y S1 y S2 son ambos provistos o usados.
  - S1 y S2 son externos y uno de ellos es el que se provee y otro es usado.

- Endpoint1 `->` Endpoint2 (Link wires) una conexión conlleva 2 elementos de especificación internos. Este tipo de conexión siempre enlaza un elemento usado especificado por Endpoint1 a otro elemento provisto especificado por Endpoint2.

  Si estas dos condiciones no se cumplen, se detectara el error en la compilación.

- Endpoint1 `<-` Endpoint2 (Link wires) es equivalente Endpoint2 `->` Endpoint1.

En los tres tipos de conexión, los dos elementos de especificación deben ser compatibles (ambos deben ser comandos, o los dos deben ser eventos, o instancias de interfaces). Si ambos son comandos o eventos deberán tener también la misma estructura de función, y si son instancias de interfaces, deben ser del mismo tipo de interfaz. En caso contrario, se avisara como error en la compilación.

Si uno de los Endpoints esta parametrizado el otro debe estarlo también, con el mismo tipo de parámetros.

Un mismo elemento de especificación puede ser conectado o enlazado varias veces:

```
configuration C {
      provides interface X;
}

implementation {
      components C1, C2;
      X = C1.X;
      X = C2.X;
}
```

En este ejemplo, un enlace múltiple conduce a señalizadotes múltiples (fan-in) para los eventos en el interfaz X y para funciones múltiples siendo ejecutadas (fan-out) cuando los comandos en la interfaz X se llaman.

Este caso de múltiples conexiones, también sucede cuando 2 configuraciones independientemente unen el mismo interfaz:

```
configuration C { }
implementation {
      components C1, C2;
      C1.Y -> C2.Y;
}
```

```
configuration D { }
implementation {
      components C3, C2;
      C3.Y -> C2.Y;
}
```

Todos los elementos de especificación externos deben ser enlazados para que no se produzca error al compilar. Sin embrago, elementos de especificación interna pueden dejarse desconectados (quizás se conecten en otra configuración o permanezcan desconectados, si los módulos tienen una implementación del evento o comando apropiado por defecto, `default`).

5.3 Conexiones implícitas

Es posible escribir `K1 <- K2.X` o `K1.X <- K2`, al igual que sucede con los otros dos tipos de conexiones (`->` y `=`)

Esta sintaxis itera a través los elementos de especificación de `K1` (o `K2`) para encontrar un elemento de especificación *Y* tal que `K1.`*Y*`<-K2.X` (tal que `K1.X<-K2.`*Y*) formara una conexión valida.

Si exactamente se puede encontrar un *Y* que cumpla esto, la conexión estará hecha, de otra forma se originarán errores de compilación. Ejemplo de un enlace simple:

```
module M1 {
provides interface StdControl;
} ...
```

```
module M2 {
     uses interface StdControl as SC;
} ...
```

```
configuration C { }
implementation {
```

```
interface X {
```

```
module M {
```

```
    command int f();                      provides interface X as P;
    event void g(int x);                  uses interface X as U;
}                                         provides command void h();
                                      } implementation { ... }

configuration C {
     provides interface X;
     provides command void h2();
}

implementation {
    components M;
    X = M.P;
    M.U -> M.P;
    h2 = M.h;
}
                    components M1, M2;
                    M2.SC -> M1;
                    }
```

La línea `M2.SC -> M1` es equivalente a `M2.SC -> M1.StdControl` en este ejemplo.

6. Concurrencia en NesC

NesC asume un modelo de ejecución que consiste en ejecución para completar las tareas (el computo en curso) y manejadores de interrupción que son señalados asíncronamente por HW.

Una planificación de NesC puede ejecutar las tareas en cualquier orden, pero se debe cumplir la regla de ejecutar-para-completar (el estándar de planificación de TinyOs sigue una política FIFO). Como las tareas no son reemplazadas y se ejecutan hasta completarse, son atómicas en lo concerniente a cada una. Pero no so atómicas con respecto a los manejadores de interrupciones.

Como este es un modelo de ejecución concurrente, los programas de NesC son susceptibles de condiciones de competición, en particular de competición entre datos en estados compartidos del programa. Por ejemplo, sus variables globales y variables de modulo (NesC no incluye asignación de memoria dinámica).

Las competiciones se evitan, o accediendo a un estado compartido solo en tareas, o solo en sentencias atómicas.

El compilador de NesC avisa de potenciales competiciones de datos en el programa en tiempo de compilación.

Formalmente se divide el código de NesC en dos partes, Código síncrono (SC) y Código asíncrono (AC). El SC es código (funciones, comandos, eventos, tareas) solo alcanzable desde las tareas. El AC es código alcanzable al menos por un manejador de interrupciones.

Aunque, el no reemplazamiento elimina las competiciones de datos entre las tareas, todavía hay una competición potencial entre SC y AC, al igual que entre AC y AC. En

general, cualquier actualización a un estado compartido que sea alcanzable desde código AC será una potencial competición de datos.

El invariante básico que se respeta en NesC es el invariante *Race-Free* (Libre de competición). Cualquier actualización del estado compartido es o solo código SC u ocurre en una sentencia atómica. El cuerpo de una función f que sea llamado desde una sentencia atómica es considerado "dentro" de la sentencia hasta que todas las llamadas a f estén "dentro" de sentencias atómicas.

NesC también reporta errores de compilación por cualquier comando o evento que sea AC y no haya sido declarado con la palabra reservada `async`. Esto asegura que el código que no fue escrito para ejecutar de forma segura en un manejador de interrupciones, no sea llamado de forma inadvertida.

Para más información sobre el lenguaje NesC consultar el manual de Referencia [6], donde se detallan en profundidad toda la sintaxis, semántica y reglas de programación.

8. Bibliografía

1. Yan Sun, Kap Luk Chan and Shankar Muthu Krishnan: *Characteristic wave detection in ECG signal using mor- phological transform*. BMC Cardiovascular Disorders 2005, 5:28.

2. Yan Sun, Kap Luk Chan and Shankar Muthu Krishnan: *ECG signal conditioning by morphological Filtering*. Com-puters in Biology and Medicine 2002. 32(6): 465 - 479.

3. C.-H. Henry Chu, E.J. Delp: *Impulsive noise suppression and background normalization of electromagnetism signals using morphological operators*, IEEE Trans.Biomed.Eng.36 (2) (1989) 262–272.

4. Base de datos de PhysioNet (the research resource for complex physiologic signals): http://www.physionet.org/

5. Bert Gyselinckx, Chris Van Hoof, Julien Ryckaert, Refet Firat Yazicioglu, Paolo Fiorini, Vladimir Leonor: *Human++: Autonomous Wireless Sensors for Body Area Networks* – IMEC

6. David Gay, Philip Levis, David Culler, Eric Brewer: *nesC 1.1 Language Reference Manual* – May 2003

7. Jason Lester Hill: *System Architecture for Wireless Sensor Networks* - University of California, Berkeley 2003

8. Nordic Semiconductor (2000), *nRF2401 Tranceiver Data Sheets,* http://www.nordicsemi.com

9. Texas Instrument, *microcontrolador MSP430x149*: http://www.ti.com/

10. Networking Issues in Wireless Sensor Networks. Deepak Ganesan: http://www.isi.edu/~weiye/pub/jpdc.pdf

11. Philip Levis and Nelson Lee: *TOSIM: A Simulator for TinyOS Networks* - Sptember 17, 2003.

12. Bertha: http://www.media.mit.edu/resenv/pubs/papers/2002-09-PushpinPervasiveWF.pdf

13. Nut/OS: http://www.proconx.com/xnut/nutos/

14. Contiki: http://www.sics.se/contiki/

15. CORMOS: http://www.ics.forth.gr/~bilas/publications/pdffiles/cormos-ewsn05-cr.pdf

16. eCos: http://www.ecoscentric.com/

17. EYESOS http://eyeos.org/es/

18. MagnetOS http://www.cs.cornell.edu/People/egs/magnetos/

19. T- Kernel: http://portal.acm.org/citation.cfm?id=1182809

20. LiteOS: http://www.sandh.com/__bbs__/tsxlite.htm

21. DUBIN, D. (1976) *Electrocardiografía práctica.* , México D. F., McGraw-Hill Interamericana.

22. TinyOS: http://www.tinyos.net/

23. Philip Levis and Nelson Lee: *TOSIM: A Simulator for TinyOS Networks* - Sptember 17, 2003

24. Victor Shnayder, Mark Hempstead, Borrong Chen, Geoff Werner Allen, and Matt Welsh: *Simulating the Power Consumption of LargeScale Sensor Network Applications*

25. Alexandru Emilian Şuşu, Andrea Acquaviva, David Atienza: *Targeting PowerTOSSIM for the SensorCubes and Online Energy Management Schemes*

26. IAR Site: http://www.iar.com

27. MSP430 IAR Embedded Workbench™ IDE User Guide for Texas Instruments' MSP430 Microcontroller Family - Part number: U430-3

28. IMEC: http://www2.imec.be/

29. Daskalov I.K. y Christov I.I. (1999): *Automatic detection of the electrocardiogram T-wave end*. Mel Biol Eng Comput, 37 (3): 348 - 353

30. Li C., Zheng C., y Thay C. F (1995): *Detection of ECG caracteristic points using wavelet transforms*. IEEE Transactions on Biomedical Engineering, 42:21 - 29