

# **Estudio y Mejoras de Memoria para un Algoritmo Basado en Diagnostico de Electrocardiogramas en Redes de Sensores Inalámbricos.**

Mónica Jiménez Antón

Profesores directores:  
David Atienza Alonso  
Marcos Sanchez-Elez Martín

Colaboradores:  
Francisco Javier Rincón Vallejos

2007 - 2008  
Master en Investigación en Informática  
Facultad de Informática  
Universidad Complutense de Madrid



## **Resumen**

La utilización de una red corporal de sensores inalámbricos es una posible alternativa para el cuidado de la salud que está en pleno desarrollo. Las limitaciones de memoria y capacidad de procesamiento son uno de los mayores problemas de los nodos que componen la red, y dificultan la creación de aplicaciones capaces de ejecutarse en ellos. Hasta ahora se optaba por enviar todos los datos recogidos, y era la estación base la que los analizaba. En este proyecto se ha conseguido integrar en una plataforma inalámbrica diseñada por IMEC, que es capaz de leer señales ECG, un algoritmo que realiza un análisis en tiempo real de los datos leídos por los sensores y reduce notablemente el número de transmisiones necesarias entre el nodo y la estación base.

La aplicación resultante es una herramienta de autodiagnóstico lo suficientemente potente como para detectar las ondas características más importantes del ECG e informar sobre los posibles problemas cardiacos del paciente.

## **Abstract**

Using a Wireless Body Sensor Networks (WBSN) is a possible alternative for healthcare that is in full swing. Limitations on memory and processing capacity are one of the biggest problems of the nodes that make up the network, and the creation of applications able to run on them is a hard work. Up to now, the nodes sent all collected data, and the base station analysed them. This project has successfully integrated an algorithm that performs a real-time analysis of read data by sensors into a wireless platform designed by IMEC, which is capable of reading signals ECG, so the number of necessary transmissions between the node and the base station has been reduced.

The resulting application is a self-diagnostic tool, powerful enough to detect the most important characteristic waves of ECG and to report on any patient's heart problems.

**Keywords: Wireless Body Sensor Networks, procesado de señales biomédicas, electrocardiogramas, ECG**



## Índice:

### Capítulo 1: Introducción

1. Redes de sensores inalámbricas .....	7
2. Sensores inalámbricos para aplicaciones biomédicas .....	8
3. Objetivos del proyecto .....	9
4. Planificación del proyecto .....	9

### Capítulo 2: Redes de Sensores Inalámbricos

5. Conceptos generales .....	11
5.1. Sistemas operativos .....	11
6. Wireless Body Sensor Nodes (WBSN) .....	13
7. Arquitectura del nodo objetivo .....	13
7.1. Hardware .....	13
7.2. Software .....	14
8. Estación base .....	22
9. TinyOS .....	24
9.1. Modelo de ejecución .....	24
9.2. Modelo de componentes .....	25
9.3. Modelo de comunicación .....	27
10. Protocolo MAC de comunicación para redes de sensores .....	29
10.1. Definición de protocolo MAC en las redes de sensores .....	29
10.2. Adaptación del protocolo MAC al WBSN .....	29
10.3. Protocolo MAC de los WBSN .....	31

### Capítulo 3: Desarrollo del algoritmo

11. Introducción al diagnóstico de ECG .....	33
12. Descripción del algoritmo original .....	37
13. Adaptación del algoritmo al nodo .....	42
14. Mejoras en Memoria .....	48
15. Estudio de los resultados .....	54
16. Conclusiones .....	56

### Apéndices

A. Instalación de TinyOS y MSP430-GCC en Windows .....	59
B. TOSSIM .....	63
C. Manual de NesC .....	69
D. Cómo portar la aplicación al nodo .....	83

Bibliografía .....	87
--------------------	----



# Capítulo 1: Introducción

## 1. Redes de sensores inalámbricas.

Las redes de sensores inalámbricos (Wireless Sensor Networks [12]) consisten en un conjunto de nodos de pequeño tamaño y de muy bajo consumo, interconectados entre sí a través de una red y, a su vez, conectados a un sistema central encargado de recopilar la información recogida por cada uno de los sensores (figura 1.1).

Esta topología en red, permite la implementación de gran cantidad de aplicaciones en las que se hace necesario recolectar datos en varios puntos. Estos pueden ser del mismo tipo (por ejemplo: niveles de contaminación en un túnel para activar los extractores) o diferentes (por ejemplo: análisis de la temperatura, humedad del suelo, etc. en un invernadero).

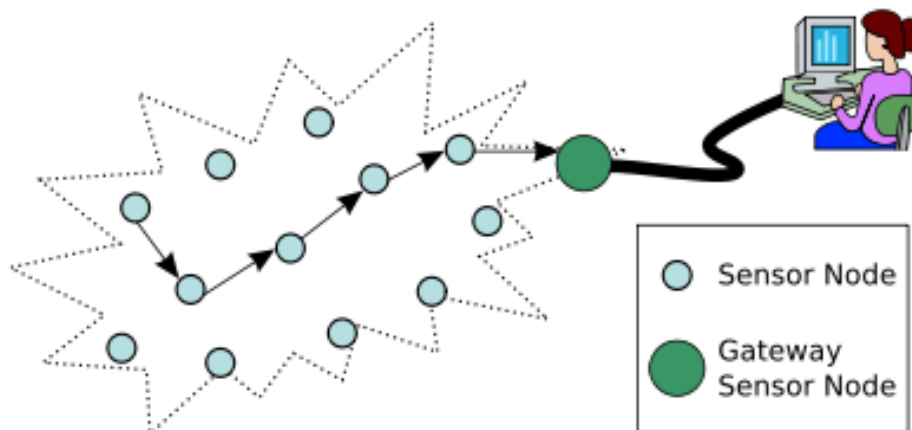


Fig. 1.1: Wireless Sensor Network.

Las redes se pueden clasificar según su modo de uso:

- Monitorización continua: los nodos miden los mismos parámetros en un área de interés, con envío periódico de la información recogida. (Como redes destinadas a la medicina, detección de Ecos, etc.)
- Monitorización basada en eventos: los nodos están monitorizando entornos de forma continuada, pero solo envían información cuando ocurre algún evento. (por ejemplo: en una central nuclear).
- Localización y seguimiento: los nodos son usados para etiquetar y localizar objetos en una zona determinada. (Como en estudios de medioambiente, para el control de la fauna de un área, etc.)
- Redes Híbridas: escenarios de acción que tienen nodos de las 3 categorías anteriores.

## 2. Sensores inalámbricos para aplicaciones biomédicas

La salud es una de las mayores preocupaciones para mejorar la calidad de vida de la población. Actualmente, es necesario acudir a tu médico u otra persona cualificada para realizar cualquier tipo de prueba médica o tratamiento. Además, para realizar monitorizaciones de un paciente durante largos periodos de tiempo, es necesario ingresarlo o utilizar una serie de aparatos de gran tamaño y difíciles de transportar, que condicionan en gran medida su rutina diaria.

Últimamente, están apareciendo nuevas tecnologías para el cuidado de la salud que permiten mejorar el estilo de vida y cubren de forma gradual las necesidades de la sociedad (por ejemplo: el implante quirúrgico de desfibriladores, dispensadores de insulina, etc.). Estas nuevas tecnologías permiten a los pacientes realizar determinados tratamientos sin necesidad de acudir a un centro hospitalario y mantener una actividad diaria normal. Se espera que para los próximos años la tecnología permita a la gente llevar sus propias redes personales colocadas en el cuerpo (debajo de la ropa o incluso en forma de pulsera) que permitan la realización de deportes y actividades cotidianas asistidas médicamente [20].

Estas redes están formadas por un conjunto de nodos inalámbricos, cada uno capaz de tomar diferentes medidas (análisis de sangre, electrocardiogramas - ECG, encefalogramas - EEG, etc.) y con suficiente inteligencia para realizar su propia tarea y permitir comunicaciones unos con otros o con una estación base (como un móvil o PDA). Se puede ver un ejemplo en la figura 2.1. La red funcionaría como un dispositivo único que puede facilitar determinados servicios a la persona que lo porte, tales como el tratamiento de enfermedades crónicas, diagnóstico médico, monitorización doméstica, entrenamiento deportivo, etc.

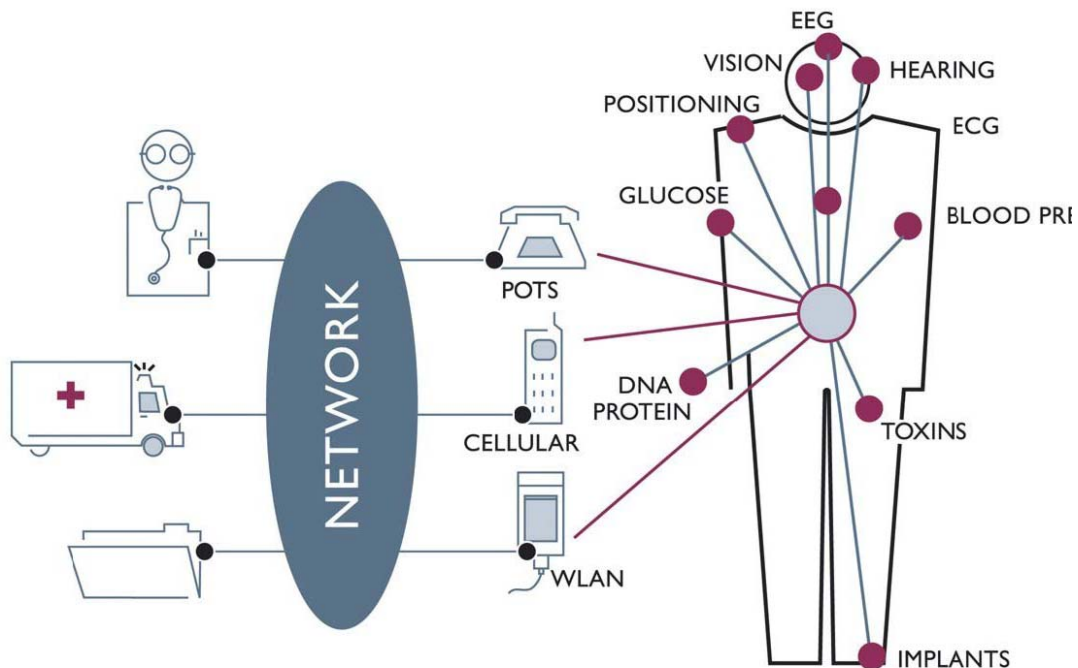


Fig. 2.1: Esquema de Red de Sensores inalámbricos para aplicaciones biomédicas.



Los principales problemas de estos nodos son la necesidad de un tamaño reducido y de gran autonomía, ya que son dispositivos ideados para ser llevados en el cuerpo, y por este motivo deben ser ligeros y discretos. Además, en determinados casos puede ser necesaria su implantación mediante una operación quirúrgica, por lo que interesa que las baterías duren el tiempo suficiente para no tener que intervenir a los pacientes cada pocos meses. Debido a esto, los nodos tienen grandes limitaciones de memoria y capacidad de procesamiento. Por otro lado, como utilizan baterías como fuente de alimentación, el consumo tiene que ser mínimo.

### **3. Objetivos del proyecto**

Dentro de los diversos tipos de nodo existentes, clasificados en función de los datos que recogen, destacan los dispositivos capaces registrar la actividad eléctrica del corazón. Éstos permiten detectar posibles cardiopatías o monitorizar y llevar el control de pacientes con enfermedades cardíacas.

Para este proyecto se ha utilizado uno de estos nodos inalámbricos, con restricciones de memoria y energía, y que son capaces de realizar electrocardiogramas o ECG. El proyecto consiste en la adaptación de un algoritmo para funcionar en este dispositivo y propone una serie de mejoras en el uso de memoria. Dicho algoritmo permite al nodo detectar ondas características en el ECG de un individuo (apartado 11) y en caso de una mala detección, envía un aviso a una estación base a través de una radio que lleva integrada el sensor.

El motivo por el cual se intenta detectar las ondas características en la señal ECG en el sensor es, que el envío de paquetes con los datos de la señal que recoge el sensor a la estación consume demasiado y no garantiza la autonomía del sensor. Por lo que se hace necesario emplear un algoritmo que analice la señal y envíe solo la información necesaria. Sin embargo, esta tarea no es trivial debido a las limitaciones del nodo y se hace necesario un estudio y análisis para reducir al mínimo el uso de memoria por parte del algoritmo.

### **4. Planificación del proyecto**

Para llevar a cabo el estudio, inicialmente se procedió a modificar un algoritmo de detección de complejos QRS perteneciente a la base de datos de PhysioNet [4]. Sin embargo, este algoritmo finalmente se descartó debido a que no era lo suficientemente completo, ya que no detectaba todas las ondas características de un electrocardiograma.

Debido a esto se tomó como referencia el algoritmo propuesto por Yan Sun [1] en el que se detectan tanto el complejo QRS como las ondas P y T (apartado 11). Se partió de un algoritmo inicial, basado en el de Yan, pero con diversas modificaciones que se explican con detalle en el apartado 13 para adaptarlo al nodo.

Se está trabajando sobre un dispositivo, con una memoria reducida, y que posee un sensor que aporta datos nuevos de forma periódica. Debido a la restricción de memoria, no es posible almacenar un gran número de datos, y es necesario analizarlos en tiempo

real, según son proporcionados. Por este motivo, la principal diferencia entre el algoritmo portado y el propuesto por Yan, es que este autor propone un algoritmo de análisis de ECG estático (con un electrocardiograma completo), mientras que nosotros necesitamos un algoritmo que analice los datos de forma dinámica, según los va proporcionando el sensor, y con el mínimo retraso posible. Posteriormente se completó el algoritmo incluyendo una serie de reglas para la detección de anomalías en el electrocardiograma.

Visto que una de las principales limitaciones de los nodos, es la cantidad de memoria disponible para las aplicaciones, una vez que se comprobó que el algoritmo funcionaba, el trabajo se centró en proponer una serie de mejoras para disminuir el uso de memoria, de forma que en las versiones finales se ha conseguido minimizar hasta casi la mitad. Esto es importante, ya no solo en este algoritmo, si no también para establecer unas pautas comunes a todos los algoritmos diseñados para funcionar en este nodo.

A continuación se explican brevemente las características de las redes de sensores y se comenta en más detalle la arquitectura y el software del nodo con el que se está trabajando (capaz de realizar electrocardiogramas).

Por otro lado se detalla en qué consiste el algoritmo de detección del que se ha partido en este proyecto, y las modificaciones que ha sufrido hasta llegar a la versión final, para poder usarse en el nodo funcionando correctamente. Por último, se incluye el estudio del uso de memoria y las conclusiones a las que se ha llegado. Para finalizar, se añade una explicación de las diversas herramientas usadas para llevarlo a cabo.

## Capítulo 2: Redes de Sensores Inalámbricos

### 5. Conceptos generales.

Ya hemos visto brevemente en la introducción (apartado 1) en qué consiste una red de sensores inalámbricos. Estas redes están compuestas por un conjunto de nodos, con unas características comunes, que se comunican entre sí o que envían sus datos a una estación base [12].

Los nodos tienen una estructura común, independientemente de su tarea, representada en la figura 5.1.

- Constan de un sensor que es el encargado de recoger los datos.
- Un procesador sencillo con una pequeña memoria, que permite realizar cálculos locales sobre los datos adquiridos por el sensor.
- La radio que se encarga de las comunicaciones inalámbricas, que normalmente se trata de una radio de baja frecuencia.
- La fuente de alimentación, que al ser inalámbricos y de pequeño tamaño suelen ser pilas o baterías recargables.

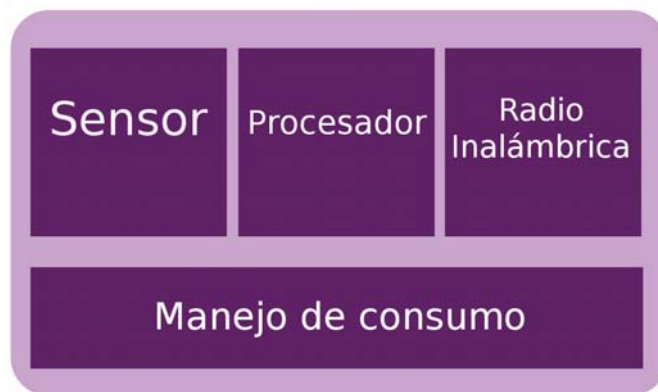


Fig. 5.1: Estructura general de la arquitectura de un nodo

Los nodos tienen que ser de tamaño reducido, especialmente aquellos destinados a ser portados por personas o animales. Por otro lado, tienen restricciones de consumo severas debido a la necesidad de que los nodos sean capaces de operar por sí mismos, durante largos periodos de tiempo, en lugares donde las fuentes de alimentación son inexistentes o que, sencillamente, necesitan usar baterías de tamaño reducido y por lo tanto de baja potencia. Otra posibilidad es el uso de *energy scavengers* que obtienen energía del medio (energía solar, energía obtenida del movimiento, etc.)

Todo esto, sumando a la reducción de los costes de producción, hace que se tenga que trabajar con nodos con grandes limitaciones en cuanto a memoria y capacidad de

procesamiento. Esto dificulta el diseño de aplicaciones capaces de operar en ellos, ajustándose a dichas limitaciones.

## 5.1. Sistemas operativos.

El uso de un sistema operativo adaptado al nodo facilita en gran medida el trabajo a la hora de especificar el comportamiento de los diferentes componentes hardware, y permite mayor flexibilidad a la hora de diseñar aplicaciones concretas, en el sentido que no hay que definir todos los procesos hardware asociados a dichas aplicaciones.

El hardware de los nodos no es muy diferente de los sistemas empotrados tradicionales y es posible usar un sistema operativo específico para sistemas empotrados, tales como eCos [16] o  $\mu$ C/OS [17]. Sin embargo, estos sistemas operativos suelen estar diseñados con propiedades de tiempo real, mientras que las redes de sensores a menudo no tienen este soporte.

El primer sistema operativo específicamente diseñado para redes de sensores inalámbricas se conoce con el nombre de TinyOS [5, 22], basado en eventos en vez de multithreading. El lenguaje asociado a este sistema se conoce con el nombre de NesC [11], que es una extensión de C, y que se explica más a fondo en el apéndice C.

Otros sistemas operativos que son adecuados para las redes de sensores, y que usan C como lenguaje de programación asociado, son Contiki, MANTIS, BTnut, SOS y Nano-RK.

- Contiki [6] está diseñado para soportar módulos de carga sobre la red y soporta la carga de ficheros ELF. Al igual que TinyOS está dirigido por eventos, pero el sistema soporta multithreading en una base por aplicación. Además, incluye protothreads que proporcionan una abstracción similar a la programación con hilos.
- MANTIS [7] y Nano-RK [8] están basados en multithreading preventivo. Con esto, las aplicaciones no necesitan ceder explícitamente el microprocesador a otros procesos. En vez de eso, el kernel divide el tiempo entre los procesos activos y decide cual de todos los que están listos para ejecutarse hace la programación de la aplicación más fácil. Nano-RK es un kernel de tiempo real y permite control de grano fino del acceso de las tareas a la CPU, las redes y el sensor.
- SOS [9] también es un sistema operativo basado en eventos. Su principal característica es el soporte de módulos cargables. Un sistema completo está formado por pequeños módulos, posiblemente en tiempo de ejecución. Por otro lado, se centra en el soporte del manejo de memoria dinámica para soportar el dinamismo inherente en la interfaz de un módulo.
- BTnut [10] está basado en multi-threading cooperativo y en código plano de C.

## 6. Wireless Body Sensor Networks (WBSN)

Las WBSN [20] consisten en un conjunto de nodos pegados al cuerpo del paciente, cada uno formado por 4 componentes: un sensor, un microprocesador, un transmisor inalámbrico y una fuente de energía.

Cada nodo está diseñado para capturar datos fisiológicos concretos (nivel de azúcar en sangre, electrocardiogramas, encefalogramas, etc.), dependiendo del funcionamiento del sensor que incorpore. Además se permite un preprocesado de estos datos y su transmisión a una estación base.

La estación base será responsable de almacenar, organizar y completar el análisis y la fusión de toda la información recolectada por los diferentes nodos.

## 7. Arquitectura del Nodo Objetivo.

En este proyecto se usa un nodo de 25 canales diseñado por IMEC [28], que pueden monitorizar 25 señales EEG/ECG, y que puede transmitir en tiempo real la información medida a la estación base. El nodo tiene una arquitectura muy elaborada (imagen 7.1) que permite portar diferentes tipos de aplicaciones de procesamiento de señales, usar diferentes componentes hardware y manejar la pila de comunicación para varios protocolos de comunicación inalámbricos.

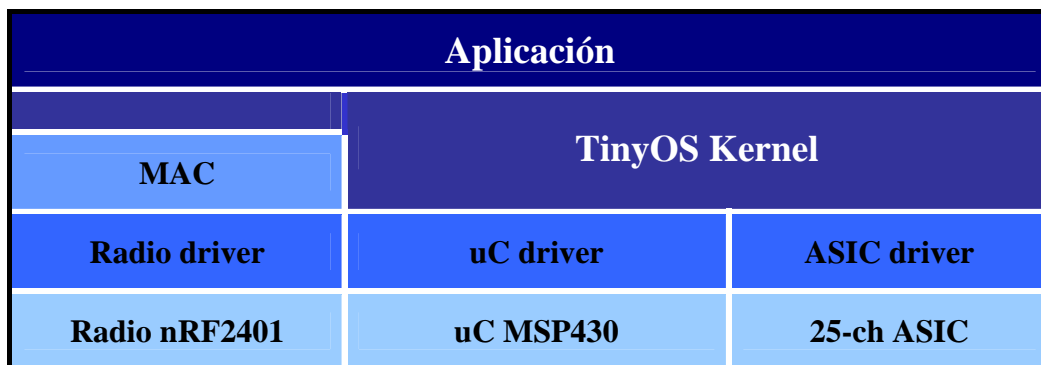


Fig. 7.1: arquitectura del WBSN de 25 canales

### 7.1. Hardware

Se sigue la estructura general de este tipo de dispositivos, formada por un sensor, CPU, un transmisor inalámbrico y fuente de alimentación.

**a) Sensor: ASIC de 25 canales.**

El sensor se trata de un ASIC de muy bajo consumo de 25 canales, capaz de captar la actividad eléctrica del cerebro (encefalograma o EEG) y el corazón (electrocardiograma o ECG), con una frecuencia máxima de muestreo de 1024Hz por canal.

**b) Procesador: MSP430x149**

La CPU esta formada por un microcontrolador de bajo consumo, **MSP430x149** de Texas Instruments [14], especialmente diseñado para dispositivos empotrados debido a su bajo consumo, con 60kB de memoria flash (ROM) y 2kB de RAM.

El MSP430 incorpora un procesador de 16bits con arquitectura RISC, algunos periféricos y un sistema de reloj flexible, interconectados usando un bus común para datos y memoria (figura 7.2).

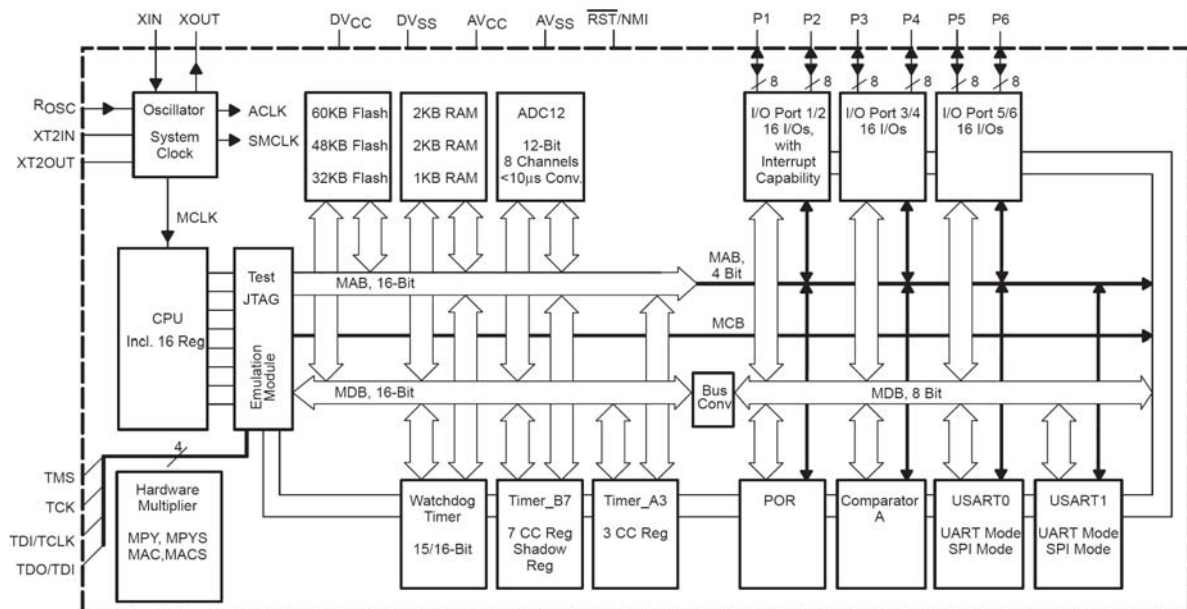


Fig. 7.2: arquitectura del MSP430

La arquitectura RISC consta de un juego de 27 instrucciones y 7 modos de direccionamiento. La arquitectura es ortogonal, cualquier instrucción puede usar cualquiera de los modos de direccionamiento. Incorpora 16 registros de 16bits totalmente accesibles, de los que R0, R1, R2 y R3 tienen funciones específicas y los registros de R4 a R15 son de propósito general.

Las instrucciones se dividen en tres tipos: de dos operandos (fuente - destino), un operando (destino) y salto. Se trata de un set muy sencillo, ya que dentro de las instrucciones de dos operandos solo se contemplan operaciones aritméticas de suma y resta, operaciones a nivel de bit y operaciones lógicas and y or, entre otras. Para realizar multiplicaciones se utiliza un multiplicador hardware independiente, es decir, se trata de un periférico añadido para la versión 149 del microcontrolador. Las operaciones están

definidas para operandos de 16 bits, por lo que no tiene soporte para operaciones con 32 bits.

El microcontrolador no tiene unidad de punto flotante, por lo que todas las operaciones con esta precisión requieren gran cantidad de soporte software, y es conveniente evitarlas.

Los 7 modos de direccionamiento se contemplan en tabla 7.1. Siete de ellos se usan en el operando fuente, y cuatro en el operando destino, y son suficientes para cubrir el espacio de direcciones completo.

Modo de direccionamiento.	Fuente	Destino	Sintaxis	Descripción
Register	✓	✓	MOV Rs, Rd	Rs --> Rd
Indexed	✓	✓	MOV X(Rn), Y(Rm)	M(x+Rn) --> M(y+Rm)
Symbolic	✓	✓	MOV ADDR, ADDR2	M(ADDR) --> M(ADDR2)
Absolute	✓	✓	MOV &ADDR, &ADDR2	M(ADDR) --> M(ADDR2)
Indirect register	✓	✗	MOV @Rn, Y(Rm)	M(Rn) --> M(y+R6)
Indirect autoincrement	✓	✗	MOV @Rn+, Rm	M(Rn) --> Rm Rn + 2 --> Rn
Immediate mode	✓	✗	MOV #X, ADDR	#45 --> M(ADDR)

Tabla 7.1: Modos de direccionamiento.

El reloj de sistema esta específicamente diseñado para dispositivos alimentados con baterías. Consta de dos relojes: un reloj de baja frecuencia auxiliar (*ACLK*) para el modo de bajo consumo stand-by y un reloj principal de alta velocidad (*MCLK*) para alto rendimiento. El sistema consta de un oscilador controlado digitalmente (*DCO*) que permite pasar de un estado a otro en menos de 6µs. Por otro lado existe un reloj sub-principal (*SMCLK*) para los periféricos.

La arquitectura de von-Neumann consta de un único espacio de direcciones compartido con registros de funciones especiales, periféricos, RAM y memoria Flash o ROM como se muestra en la figura 7.3. El tamaño de la RAM y la ROM varía de unas versiones del microcontrolador a otras, este caso se trata del MSP430x149, que contiene 60kB de ROM y 2kB de RAM.

La ROM puede usarse para almacenar tanto código como datos. Las palabras pueden ser almacenadas y usadas en esta memoria sin necesidad de copiarlas a la RAM.

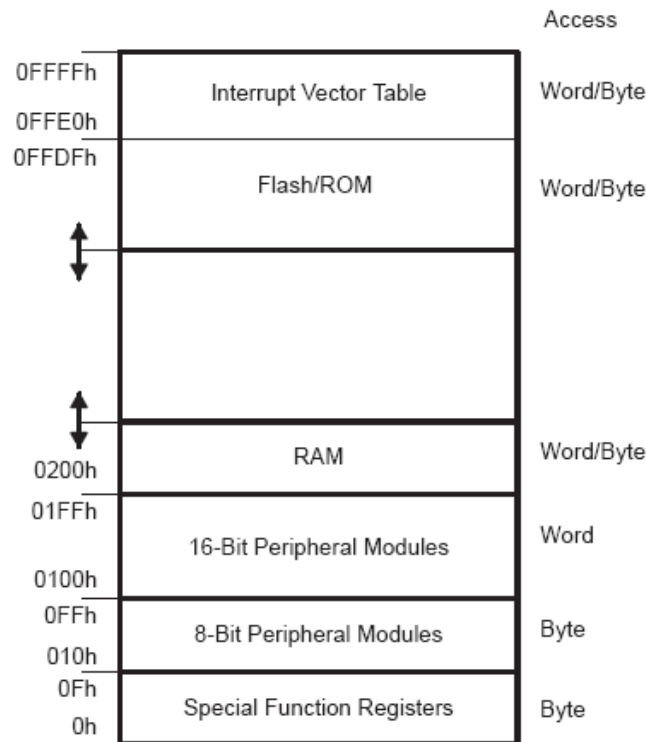


Fig. 7.3: Mapa de memoria del MSP430

La familia de este procesador esta especialmente diseñada para aplicaciones con muy bajo consumo y usa diferentes modos de operación configurables por software teniendo en cuenta las necesidades del sistema. Tiene un modo activo y 5 modos de bajo consumo. En el caso de que llegue una interrupción, puede despertar al dispositivo de cualquiera de los 5 modos de bajo consumo, resolver la interrupción y volver al modo de bajo consumo en el que estaba.

- Modo activo (AM)
  - La CPU y todos los relojes están activos.
- Low-power mode 0 (LPM0)
  - CPU desconectada.
  - ACLK y SMCLK activos. MCLK desconectado.
- Low-power mode 1 (LPM1)
  - CPU desconectada.
  - ACLK y SMCLK activos. MCLK desconectado.
  - Generador DC desconectado si el DCO no se usa en el modo activo.
- Low-power mode 2 (LPM2)
  - CPU desconectada.
  - MCLK y SMCLK desconectados y ACLK activo.
- Low-power mode 3 (LPM3)
  - CPU desconectada.
  - MCLK y SMCLK desconectados y ACLK activo.
  - Generador DC desconectado
- Low-power mode 4 (LPM4)
  - CPU y todos los relojes están inactivos.



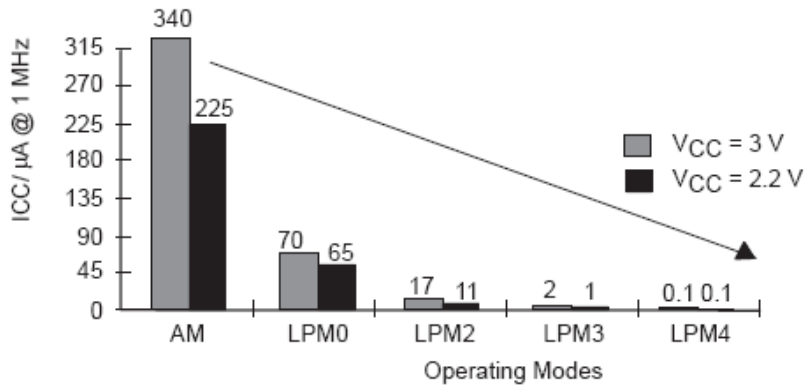


Fig. 7.4: Decremento del consumo para los diferentes modos en el MSP430

Observando la gráfica de la figura 7.4 se puede ver el bajo consumo de este microcontrolador. 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 (LPM4) tendrá 0.1µA. Según se aumenta la frecuencia de reloj se aumenta la energía necesaria y por tanto el consumo. El consumo máximo se alcanza en estado activo con el reloj funcionando a 8MHz y un voltaje de 3.6V (figura 7.5).

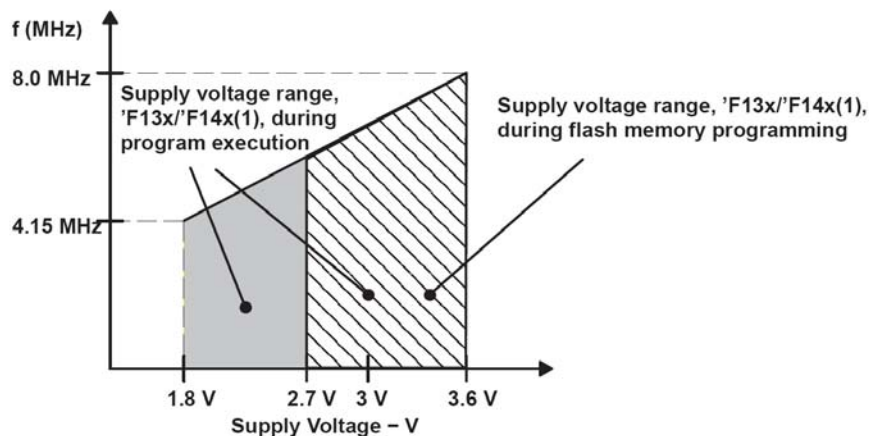


Fig. 7.5: Frecuencia vs. Voltaje de entrada del MSP430

Por último comentar que el MSP430 consta de una serie de periféricos. El más útil para el sensor EEG/ECG es un convertidor analógico-digital de 12 bits, ya que facilita el procesamiento de señales biomédicas.

Para más información del microcontrolador del nodo se pueden consultar la página Web de Texas Instrument [14].

### c) Transmisor: Nordic nRF2401.

Para la unidad de comunicación se usa un transmisor-receptor **Nordic nRF2401** [13], un chip de bajo consumo con un ancho de banda de 2.4 a 2.5GHz. Este transmisor esta

formado por un sintetizador de frecuencia totalmente integrado, un amplificador de potencia, un oscilador de cristal y un modulador.

Existe un modo de configuración, durante el que se carga una palabra de configuración de 15bytes mediante una interfaz de 3 conexiones (CE, CLK1, DATA). La funcionalidad del dispositivo en estos modos se decide por el contenido de dicha palabra de configuración, y la tasa de datos se decide por la velocidad del microcontrolador asociado.

La radio envía paquetes de 33bytes, cuyo formato se indica en la figura 7.6.



Fig. 7.6: Formato del paquete enviado por el transmisor nRF2401

Donde:

- ADDR es la dirección destino,
- PAYLOAD: datos,
- CRC: comprobación de redundancia cíclica.

El consumo normal es muy bajo, 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.

Además tiene cinco modos de consumo diferentes:

- **ShockBurst (modo activo).**

Cuando se opera en este modo, se gana acceso a grandes tasas de datos (máximo 1Mbps), ofrecidas por un ancho de banda de 2.4GHz, sin la necesidad de un microcontrolador para el procesamiento de datos, ya que el chip de la radio se encarga también de chequear el CRC del paquete recibido. De esta forma se consigue una gran velocidad de transferencia con un bajo rendimiento del procesador.

La importancia de este modo de comunicación radica en que no es necesario sincronizar el microcontrolador con la radio. Esto quiere decir, que el microcontrolador puede enviar datos para ser enviados a una frecuencia diferente a la que la radio envía dichos paquetes (figura 7.7). 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 7.8).

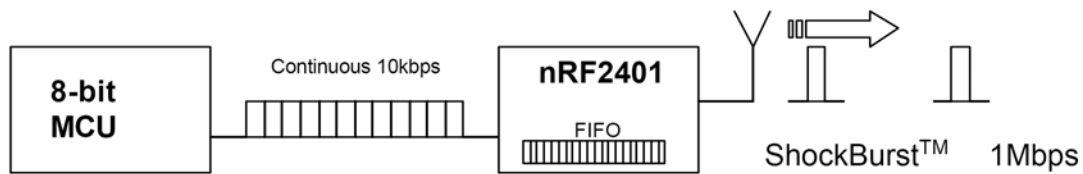


Fig. 7.7: Sincronización de los datos entre el microcontrolador y la radio

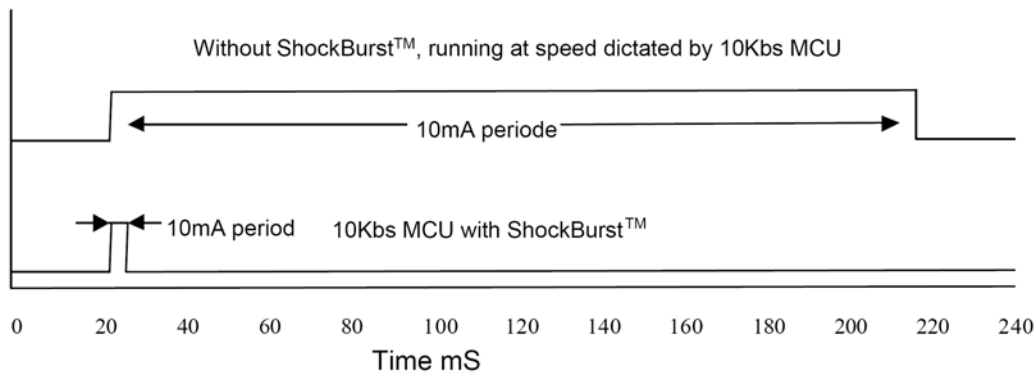


Fig. 7.8: Consumo total sin ShockBurst y con ShockBurst

#### Trasmisión:

1. Cuando el microcontrolador tiene un dato para enviar, activa la señal CE, esto configura el NRF2401 en procesamiento de datos a bordo.
2. 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 entre 0 y 1Mbps.
3. El microcontrolador establece CE a cero, y esto activa la transmisión ShockBurst.
  - a. Se enciende la radio.
  - b. Se completa el paquete (Se añade preámbulo y se calcula el CRC).
  - c. Se transmiten los datos a alta velocidad (250 kbps o 1 Mbps configurada por el usuario).
  - d. El transmisor vuelve al modo stand-by cuando termina.

Esta idea es equivalente a la inversa para la recepción de datos.

#### Recepción:

1. La dirección y el tamaño de los datos de los paquetes de entrada se establece cuando se activa el modo de recepción (RX) de ShockBurst.
2. Dicho modo RX se configura activando la señal CE.
3. Cuando se recibe un paquete válido (dirección y CRC correctos) se queda solo con el campo de datos.
4. Entonces se notifica al microcontrolador (se interrumpe), estableciendo el pin DR1 a alta.

5. El microcontrolador podría (o no) fijar el deshabilitar la radio (Power Down Mode), desactivando la señal CE.
6. El microcontrolador ficha los datos en una tasa adecuada (ejemplo: 10Kbps).
7. Cuando se han recuperado 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 empieza la secuencia desde el principio.

- **Direct Mode (modo activo).**

El nRF2401 funciona como un receptor tradicional, donde los datos deben estar  $1\text{Mbps} \pm 200\text{ppm}$  (o 250Kbps para una configuración de tasa de datos baja), para que el receptor detecte las señales. Es decir, el microcontrolador debe estar sincronizado con la radio, para enviarle/recibir los datos a la misma velocidad que esta los transmite/recibe.

Transmisión:

1. Cuando el microcontrolador tiene un dato activa CE.
2. La radio se activa inmediatamente y tras 200us de tiempo establecimiento, los datos se cargarán directamente.
3. Todas las partes del protocolo de radio deben ser implementadas en firmware del microcontrolador (preámbulo, dirección y CRC).

Recepción:

1. Una vez que el nRF2401 ha sido configurado y encendido en modo de recepción (CE a alta), DATA empezara a conmutar debido a ruido.
2. El reloj también empezara a conmutar ya que el transmisor esta intentando bloquear el flujo de datos entrantes.
3. Una vez que llega un preámbulo válido, CLK1 y Data localizaran y seguirán la señal entrante y el paquete llegará al puerto DATA con la misma velocidad que fue transmitido.
4. Para encender el demodulador para 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.
5. En este modo, se dispone de la señal DR (no data Reddy). Las direcciones y el CRC también deben establecerse en el microcontrolador.

- **Stand-By Mode (Modo de bajo consumo)**

Se usa para minimizar el consumo medio mientras se mantienen cortos tiempos de inicio. El consumo medio depende de la frecuencia del oscilador ( $12\mu\text{A}$  con 4MHz,  $32\mu\text{A}$  con 16 MHz).

- **Power Down Mode (modo de bajo consumo).**

Se desconecta el transmisor que pasa a consumir el mínimo posible, menos de  $1\mu\text{A}$ , aumentando el tiempo de vida de la batería.

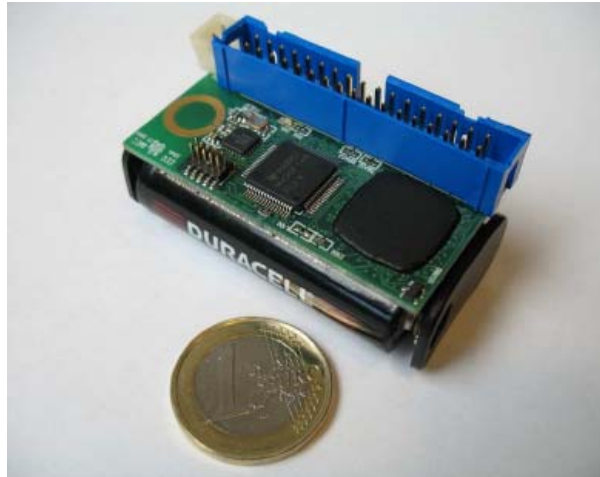


Fig. 7.9: Imagen del Wireless Body Sensor Node

#### **d) Fuente de alimentación**

Por ultimo, comentar la fuente de alimentación. El sistema completo necesita un aporte de energía en un rango de  $2.7\text{V} - 3.3\text{V}$ , por lo que será suficiente con usar dos baterías alcalinas AA.

Existen otras posibilidades, como por ejemplo, el uso de *energy scavengers*, que son dispositivos que transforman la energía ambiental a energía eléctrica. Hay muchos tipos, pero para este tipo de sensores son útiles los dispositivos que son capaces de transformar el movimiento o la diferencia térmica entre el exterior y el individuo que porta el nodo en electricidad.

## **7.2. Software**

Para la parte software se ha definido un módulo separado para cada componente hardware (sensor, microcontrolador y radio). Para soportarlo, se incluye en el nodo un sencillo sistema operativo basado en eventos llamado TinyOS [5, 22], que ya se ha nombrado anteriormente, y que está especialmente diseñado para redes de sensores inalámbricos, del que se hablará con más profundidad en el apartado 9. Gracias a esto, es posible portar aplicaciones para el procesamiento de la señal fácilmente, usando los driver provistos por TinyOs para acceder a los diferentes bloques hardware. Además, la abstracción de dichos bloques hardware hace posible modificarlos o reemplazarlos sin perjudicar al resto del sistema.

## 8. Estación base

La estación base o receptor de datos asociada a la WBSN tiene la misma arquitectura que los nodos normales pero en lugar de sensores tiene un conector USB para poder ser conecta a un PC. Para más información de cómo instalarla se necesita consultar el apéndice D.

El nodo envía al receptor paquetes de 33bytes con 18 bytes de datos. Cuando el paquete llega a la estación base, se le quita el preámbulo y el CRC (comprobando que se ha transmitido correctamente).



Fig. 8.1: Formato del paquete mostrado en el programa Terminal

Para visualizar los paquetes recibidos se usa el programa *Terminal*. Los paquetes se muestran según el formato de la figura 8.1, donde 7E son dos bytes añadidos que indican el comienzo y el final de paquete.

En la figura 8.2 se muestra la pantalla principal de éste programa. Se trata de un interfaz muy sencillo, donde se pueden variar directamente los parámetros de recepción y se visualizan los paquetes recibidos. En el panel de mayor tamaño, se visualizan los paquetes recibidos, en formato hexadecimal. Para distinguir entre un paquete y otro, conviene recordar que los mensajes comienzan y acaban con dos bytes reservados. El programa incluye la posibilidad de traducir de hexadecimal a binario y decimal, que se muestra en los paneles de la derecha.

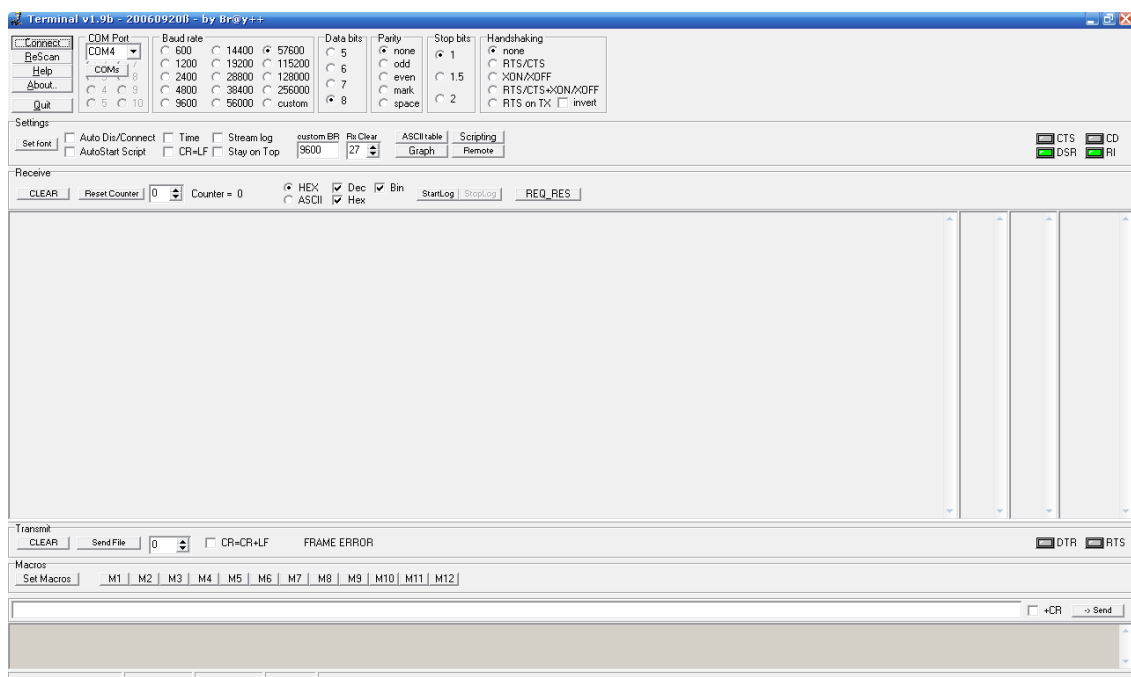


Fig. 8.2: Pantalla principal del Terminal.

Antes de conectar (connect) la estación base para que comience a recibir, es necesario configurar los parámetros de recepción en el programa (imagen 8.3). El más importante, que puede variar en otras versiones del nodo, es la velocidad de recepción o *Baud rate*, establecida por el fabricante a 57600. El resto de parámetros se dejan por defecto con *Data bits* a 8, *Parity* a none, *stop bits* a 1 y *Handshaking* a none.

*Com Port* es el puerto al que se conecta la estación base dentro de nuestro ordenador. Este debe ser siempre el mismo, y coincide con el puerto al que estaba conectada la estación en el momento de su instalación.

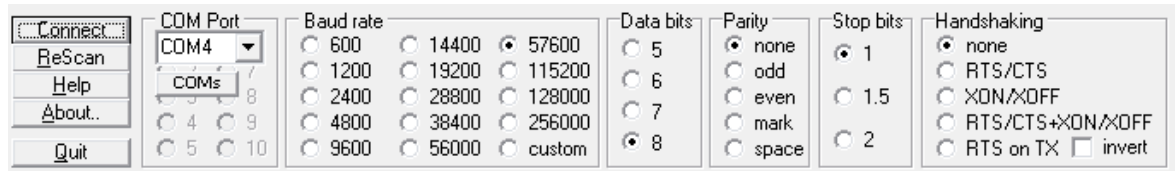


Fig. 8.3: Configuración del programa.

## 9. TinyOS

TinyOS [5, 22] es un SO de código abierto, desarrollado en la universidad de Berkeley como sistema base para la construcción de aplicaciones en sistemas empujados, específicamente redes de sensores inalámbricos (WSN).

Está programado en un meta-lenguaje derivado de C, cuyo nombre es NesC [11] y que es el lenguaje en el que hay que escribir las aplicaciones que se van a cargar en los nodos con dicho sistema operativo y que se explica con más detalle en el apéndice C.

Es importante mencionar que el proyecto se inició sobre BSD (Unix), por lo cual para el funcionamiento en Windows se hace necesario la instalación de *Cygwin* [29], un simulador de plataformas unix, siendo necesario tener conocimientos básicos acerca de este entorno. Además muchas de las herramientas disponibles se encuentran desarrolladas para plataformas JAVA, por tanto también es necesaria su instalación. Se incluye una explicación más detallada en el Apéndice A.

El diseño de TinyOS se realizó para responder a las características y necesidades de las redes de sensores, tales como reducido tamaño de memoria, bajo consumo de energía, operaciones de concurrencia intensiva, diversidad en diseños y usos, y finalmente operaciones robustas para facilitar el desarrollo confiable de aplicaciones. Además se encuentra optimizado en términos de uso de memoria y eficiencia de energía.

Podemos enumerar las siguientes características de TinyOS:

- El kernel del sistema operativo 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.
- Concurrencia de tareas y basada en eventos.
- Implementación en NesC (Apéndice C).

### 9.1. Modelo de ejecución

El diseño del Kernel de TinyOS 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 y 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, sin usar votaciones ni otros mecanismos para averiguarlo.

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

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

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

## **9.2. Modelo de componentes**

TinyOS posee un modelo de programación característico de los sistemas empujados: el modelo de componentes. Tal como se ha mencionado TinyOS está escrito en NesC, lenguaje que surge para facilitar el desarrollo de este tipo de aplicaciones. A continuación se presentan las ideas de este modelo.

- Especificación del sistema: TinyOS se encuentra construido sobre un conjunto de componentes, las cuales proveen la base para la creación de aplicaciones. Las aplicaciones serán una lista de componentes y la especificación de las interconexiones entre ellas.
- Concurrencia en tareas y en eventos: Las dos fuentes de concurrencia en TinyOS son las tareas y los eventos. Las componentes entregan tareas al planificador, siendo el retorno de éste de forma inmediata, aplazando el cómputo hasta que el planificador ejecute la tarea. Las componentes pueden realizar tareas siempre y cuando los requerimientos de tiempo no sean críticos. Para asegurar que el tiempo de espera no sea muy largo, se recomienda programar tareas cortas, y en caso de necesitar procesamientos mayores, se recomienda dividirlo en múltiples tareas. 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. Concretamente, 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 señalar a través de un evento, el éxito de la operación.

El modelo de componentes permite la fácil migración a otro hardware, dada la abstracción que se logra con el modelo de manejo de eventos. Además, la migración es particularmente importante en las redes de sensores, ya que constantemente aparecen nuevas tecnologías.

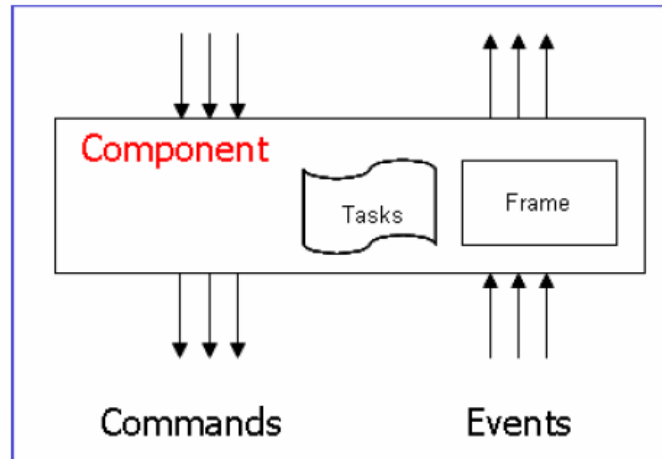


Fig. 9.1: Componente

Cada componente está formada por cuatro elementos (imagen 9.1).

- Manejador de comandos: Los comandos son peticiones hechas a componentes de capas inferiores. Estos generalmente son solicitados para ejecutar alguna operación. Existen dos posibles casos de uso de los comandos.
  - El primero es para operaciones bifase, donde los comandos retornan inmediatamente, no generando bloqueos por la espera de la ejecución. Es decir, una vez que realizan la petición, el planificador será el encargado de ejecutar lo solicitado, generando un evento que indique el fin de la operación.
  - El segundo es para una operación no bifase, donde ésta se realizará completamente, y por tanto no habrá evento retornado.
- 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 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. Debido a esta implementación, las tareas se ejecutan secuencialmente y no deben ser excesivamente largas.

### **Tipos de Componentes.**

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

- Abstracciones de Hardware: Mapean el hardware físico en el modelo de componentes de TinyOS.

- **Hardware Sintético:** simulan el comportamiento del hardware avanzado. Conceptualmente, esta componente es una máquina de estado que podría ser directamente modelada en el hardware. Desde el punto de vista de los niveles superiores, esta componente provee una interfaz, funcionalmente muy similar a una componente de abstracción de hardware.
- **Componente de alto nivel:** Realizan el control, enrutamientos y toda la transferencia de datos. Además, las componentes que realizan cálculos sobre los datos o su agregación, entran en esta categoría.

En la figura 9.2 se puede ver un ejemplo de un sistema completo, formado por componentes de diferentes tipos. Las componentes marcadas en rojo, corresponden a abstracciones hardware, que exportan comandos para manipular los pines de la placa. El componente Radio Byte se trata de hardware sintético, que se encarga de la codificación y decodificación de los datos recibidos de las componentes inferior y superior. Y por último, el resto de componentes marcadas en azul son componentes de alto nivel.

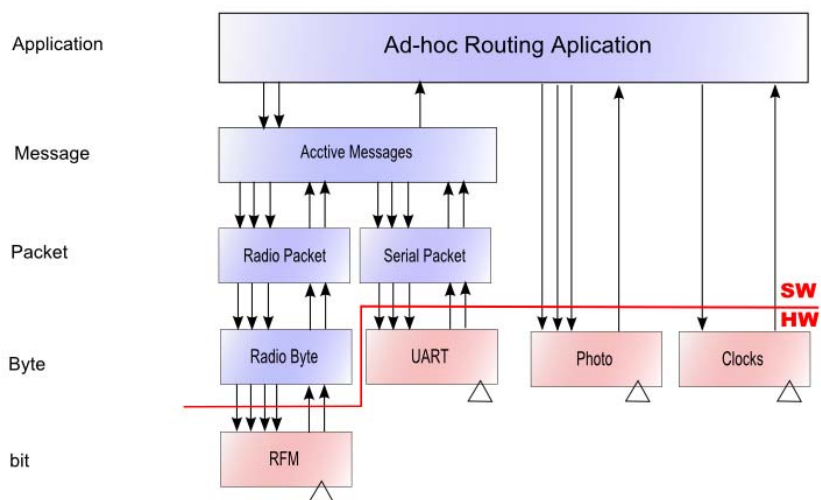


Fig. 9.2: Ejemplo de modelo de componentes.

### 9.3. Modelo de comunicación.

El modelo de comunicación definido en TinyOS usa la metodología Active Message (AM). Esto es una forma de comunicación, basada en mensajes, usada en sistemas paralelos y distribuidos.

Cada Mensaje activo contiene el nombre del manejador a ser invocado en el nodo a su llegada y un dato a pasar como argumento a dicho manejador.

Para su funcionamiento, el que envía debe declarar un buffer en el frame, para así colocar el nombre del manejador, luego solicitar el envío y esperar que llegue la respuesta de realizado. El receptor entonces se encarga de invocar el correspondiente

manejador de eventos. Una vez invocado, el manejador extrae el mensaje de la red, integra el dato en el cómputo y envía un mensaje de respuesta. De esta forma no se generan bloqueos o esperas en el receptor, el comportamiento es similar a que ocurriese un evento, y el almacenamiento es simple.

La comunicación entre la radio y las aplicaciones se hace a través del intercambio de buffers, que las aplicaciones devuelven vacíos a la radio.

## **10. Protocolo MAC de comunicación para redes de sensores**

### **10.1. Definición de protocolo MAC en las redes de sensores.**

Un protocolo MAC sirve 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 capa MAC para enviar y recibir paquetes, sincronizar sus operaciones, etc.

Su principal función es controlar el acceso al medio compartido, que en el caso de los sensores se trata de un canal de radio. El protocolo debe evitar las interferencias entre transmisiones, minimizando el efecto de las colisiones.

### **10.2. Adaptación del protocolo MAC al WBSN**

En TinyOs hay una arquitectura software disponible para el microcontrolador TI MSP430 [14], sin embargo no es el caso del modulo de radio usado en el nodo, el Nordic nRF2401 [13]. El colegio universitario de Londres produjo una HPL (Hardware Presentation Layer) que fue directamente acoplada con una conexión basada en un protocolo MAC.

Pero este diseño no se adapta al flujo de trabajo porque no permite cambiar de un protocolo MAC a otro, así que es necesaria una arquitectura más modular.

Objetivos:

- Proveer una comunidad científica para el nRF2401.
- Conseguir la portabilidad de los protocolos MAC entre las diferentes plataformas.
- Conseguir la comparación entre protocolos MAC desarrollados por grupos de trabajo diferentes para manejar la optimización de consumo en la capa MAC.

#### **10.2.1. Arquitectura del Driver.**

En la figura 10.1 se representa un esquema de la estructura del driver de la radio del nodo (nRF2401) para adaptarlo a TinyOS para que use un protocolo MAC de comunicación.

Los bloques verdes son configuraciones usadas por TinyOS para conectar los diferentes componentes, representados por 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.

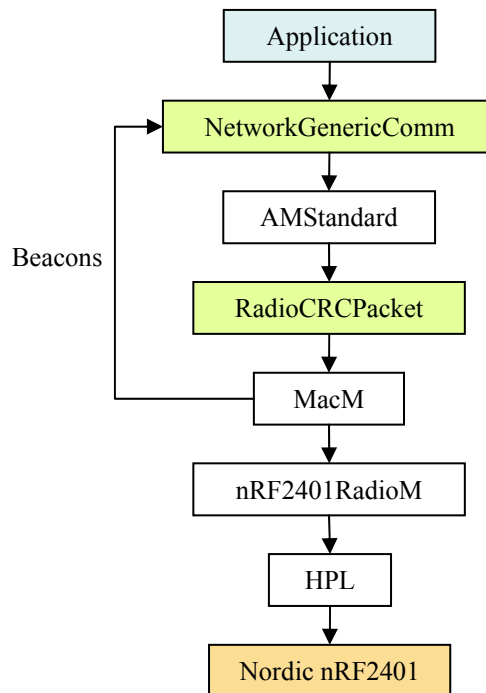


Fig. 10.1: estructura del driver del nRF2401 para TinyOS 1.0.

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 está 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 desarrolla 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. Esta capa ha sido desarrollada por el Colegio de Londres.

En la tabla 10.2 se resume la contribución a los diferentes módulos de los grupos de investigación involucrados en el desarrollo del driver.

Component	TinyOS standard	IMEC-NL	EPFL	UCL
NetworkGenericComm	X			
AMStandard	X			
RadioCRCPacket	X			
MacM		X <sup>1</sup>	X	
nRF2401RadioM		X <sup>1</sup>		X
HPL				X

Tabla 10.2: Contribución a los diferentes módulos.

### 10.3. Protocolo MAC del nodo.

Como se vio en el apartado 7, el sensor de 25 canales EEG/ECG que se usa para este proyecto utiliza un protocolo MAC para realizar las comunicaciones entre nodos. En este caso, se ha elegido un protocolo TDMA (Acceso Múltiple por división de tiempo) que permite a varios usuarios compartir el mismo canal dividiendo el tiempo en diferentes *slots*.

Cada nodo tiene un *slot* asignado, si un nodo tiene un dato que mandar, éste será enviado en ese *slot*. Esto permite que varios nodos compartan el mismo medio de transmisión (como por ejemplo, un canal de radiofrecuencia).

Se han implementado dos versiones del protocolo, la estática y la dinámica.

#### 10.3.1. 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 a las peticiones de los nodos en el siguiente mensaje de control.

Por otro lado, cuando un nodo nuevo se enciende, envía una petición de slot, que se reenviará si la estación base no le ha asignado ningún slot a dicho nodo pasado un tiempo.

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

La ventaja de usar un protocolo estático es que el número de *slots* es un número fijo y conocido. La desventaja es que el número de nodos no se puede incrementar/disminuir de forma dinámica.

<sup>1</sup> Reorganización de la versión original.

Se puede ver este comportamiento en la imagen 10.2. En el primer ciclo TDMA, el nodo  $i$  manda 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  $j$  realizará la misma petición, y se le asignará el segundo slot, y así sucesivamente hasta completar número de slots definidos inicialmente.

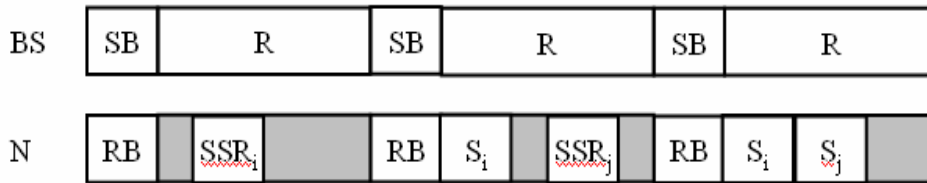


Fig. 10.2: Mensajes de control en el TDMA Estático (BS: estación base)

### 10.3.2. Protocolo TDMA Dinámico.

Está propuesto para adaptarse al número de nodos de la red de forma dinámica.

Al comienzo, la estación base manda un mensaje de control y a continuación deja un slot vacío. Este slot se usa solo para las peticiones de los nodos. Cuando un nodo desea pedir un slot, manda su petición en este slot vacío. La estación base le indicará el slot asignado a dicho nodo en el siguiente mensaje de control.

El slot vacío es siempre el siguiente slot después del mensaje de control, y los slots dedicados a transmisión de los nodos 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 vacío 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.

Fijándonos en la figura 10.3, en el segundo ciclo TDMA, el nodo  $i$  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  $i$ , tras el slot vacío. El resto de nodos realizan las peticiones de la misma manera.



Fig. 10.3: Mensajes de control en el TDMA Dinámico (BS: estación base).



## Capítulo 3: Desarrollo del algoritmo.

### 11. Introducción al diagnóstico de ECG

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

Dentro de un electrocardiograma se distinguen un conjunto de ondas características y nos aportan información sobre el estado de salud del individuo (figura 11.1).

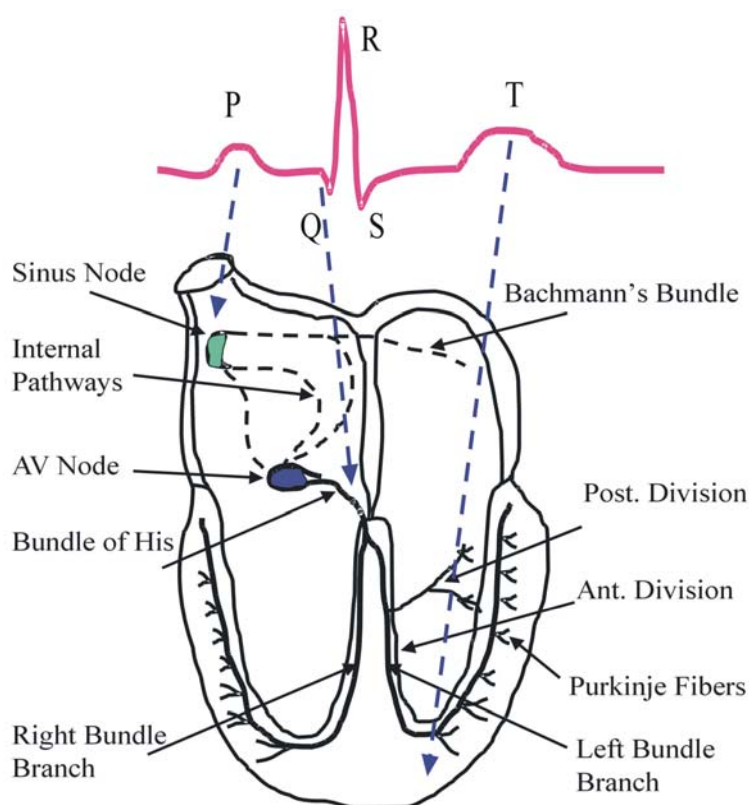


Fig. 11.1: Forma característica de un ECG y las partes del corazón que las originan.

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

- Onda P: Indica la despolarización y contracción de las aurículas. El onset y offset de P son los puntos que marcan su inicio y final.

- Complejo QRS: Indica la despolarización y contracción ventricular. Éste se divide a su vez:
  - Onda Q (Q wave): siempre hacia abajo y seguida de R.
  - Onda S (S wave): siempre hacia abajo y precedida de R.
- Onda T: Indica la repolarización ventricular (la repolarización auricular queda enmascarada por el complejo QRS). Los puntos de inicio y final se llaman onset y offset de T.

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. En la tabla 11.1 se pueden consultar estas tasas.

	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. 11.1: Frecuencias Cardiacas normales.

Teniendo en cuenta estos valores de la frecuencia y aplicando una serie de reglas que se deberían cumplir en un individuo sano podemos detectar anomalías asociadas a problemas cardiacos comunes.

Estas reglas consisten en una serie de propiedades que deben cumplir los puntos característicos del ECG (imagen 11.2).

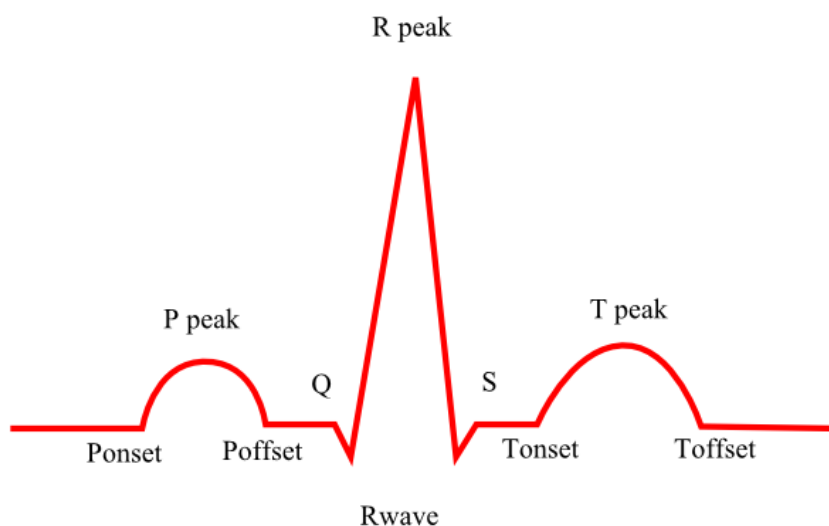


Fig. 11.2: Complejo QRS y ondas P y T

### Reglas de Normalidad:

- 1) La distancia desde Q a S debe de ser  $\leq 0,10$  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:

$$QTc = \frac{\text{Intervalo de Q a Toffset}}{\sqrt{\text{Intervalo RR anterior}}}$$

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

La corrección del intervalo QT (QTc) debe estar comprendida dentro de los valores normales que se muestran en la tabla 11.2.

Frecuencia Cardiaca/min	Intervalo RR (s)	QTc (s)	Límites normales
40	1.5	0.46	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

Tabla. 11.2 Criterios de normalidad del ECG dependiendo del intervalo RR anterior y de la corrección del intervalo QT

En la tabla 11.3 podemos ver los problemas más comunes asociados con el incumplimiento de las reglas anteriores. Es necesario tener un conocimiento avanzado de la materia para comprender las posibles patologías detalladas.

	<b>Posible Problema Cardiac</b>
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 (es mayor)	La repolarización ventricular se ha ralentizado por causas adquiridas o congénitas. Relacionado con la aparición de arritmias.
Regla 5 (es menor)	Problema normalmente relacionado con el uso de algunas medicinas, hipercalcemia o hiperpotasemia.

Tabla. 11.3 Problemas derivados del incumplimiento de las reglas de normalidad.

## 12. Descripción del algoritmo original

La detección de las principales ondas características, descritas en el apartado 11, es una de las tareas esenciales en el análisis de ECG. La dificultad de la detección de ondas radica en la oscilación de la línea base y la morfología irregular de curvas. Por este motivo, se necesitan técnicas sofisticadas para mejorar la exactitud de las detecciones.

Para realizar el estudio de memoria, se ha elegido un algoritmo propuesto en el artículo de Yan Sun [1], el cual detecta ondas características de ECG usando una transformada morfológica derivativa (MMD) y buscando en ella los puntos característicos eligiendo máximos y mínimos locales que superen unos umbrales determinados. Este algoritmo está probado, con buenos resultados, para un conjunto de electrocardiogramas de la base de datos *MIT-BIH arrhythmia* y *QT* de PhysioNet [4].

A continuación se detallan los pasos de dicho algoritmo.

### 1) Preprocesado de la señal ECG usando filtrado morfológico (MMF).

En este paso se aplican una serie de operaciones para reducir el ruido y la corrección de línea de la señal original.

$$F_b = F_o \circ B_o \bullet B_c$$
$$F = \frac{1}{2} ( (F_o - F_b) \oplus B_1 \ominus B_2 + (F_o - F_b) \ominus B_1 \oplus B_2 )$$

Donde:

- $F_o$  es la señal original recogida por el sensor.
- $F_b$  es la señal tras la corrección de línea.
- $F$  es la señal sin ruido
- $B_1$ ,  $B_2$ ,  $B_o$  y  $B_c$  son los elementos estructurales que se seleccionan basándose en las propiedades de las ondas características de ECG.

Para el algoritmo se han tomado  $B_o$  y  $B_c$  como dos segmentos horizontales de amplitud cero cuyas longitudes son:

- $L_o = 0,2 * \text{Freq}$  (para  $\text{Freq} = 200 \text{ Hz}$ ,  $L_o$  sería 40).
- $L_c = 1,5 * L_o$  (siendo  $L_o = 40$ ,  $L_c$  tendría un valor de 60).

$B_1$  y  $B_2$  se seleccionan según la Frecuencia de muestreo. En el algoritmo son vectores de longitud 5 con los siguientes valores:

$$B_1 = \{0, 1, 5, 1, 0\} \text{ y } B_2 = \{0, 0, 0, 0, 0\}$$

Las operaciones usadas en la corrección de línea son la apertura y el cierre, estas se calculan a partir de la erosión y la dilatación:

$$\text{Apertura } (\circ) \quad f \circ B = f \ominus B \oplus B$$
$$\text{Cierre } (\bullet) \quad f \bullet B = f \oplus B \ominus B$$

Las operaciones morfológicas de erosión  $\ominus$  y dilatación  $\oplus$  se calculan mediante las siguientes fórmulas:

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

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

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

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

En la figura 12.1 podemos ver un ejemplo de señal filtrada. 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.



Fig. 12.1: ECG con ruido y ECG filtrado usando filtrado morfológico.

## 2) Transformada Morfológica Multiescala (MMT).

En este paso se aplica la Transformada Morfológica Multiescala (MMT) descrita con más profundidad en los artículos de las referencias [1] y [2].

Consiste en calcular cada dato a partir los valores máximo y mínimo que le rodean, dentro de una ventana de  $2s+1$  datos, aplicando la siguiente fórmula.

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

Donde, “s” debe cumplir la siguiente condición:

$$W \text{ (Anchura de la onda característica)} * F_s \text{ (Frecuencia Muestreo)} > s$$

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, que tiene una frecuencia de muestreo de 360Hz, y un valor de 15 para la base de datos QT de PhysioNet, que tiene una frecuencia de 250Hz.



Fig. 12.2: Señal sin ruido y señal transformada mediante la transformada morfológica multiescala.

En la figura 12.2 se puede ver un ejemplo de señal transformada mediante la transformada morfológica multiestacala.

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

Para determinar los puntos característicos del ECG es necesario seleccionar unos umbrales  $Thr$  y  $Thf$  a partir de un método de Thresholding adaptativo realizado sobre el histograma de la señal transformada en el Paso 2.

Un histograma de la transformada consiste analizar la frecuencia con la que aparece cada valor. Es decir, los valores de la transformada estarán en un rango definido entre el valor máximo y el mínimo de dicha señal, y hacer su histograma consiste en analizar el número de veces que se repite cada valor dentro de la señal.

El Thresholding adaptativo se trata de un método muy usado en el tratamiento de imágenes, este consiste en definir dos umbrales a partir del histograma de una imagen. En este caso se aplica el mismo método, pero para los valores absolutos de la señal transformada. Los valores de  $Thr$  y  $Thf$  los dan los dos valles entre picos en el histograma.

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

Coincide con los mínimos locales de la transformada, cuya amplitud sea menor que el valor de  $-Thr$ .

### **5) Detección de la onda R (Rwave)**

Para cada Rpeak detectado, el primer máximo local a la izquierda que supere el umbral  $Thf$  será el inicio de R wave, y el primer máximo local a la derecha que supere el umbral  $Thf$  será detectado como el final.

### **6) Detección de la onda Q**

Coincide con el primer mínimo local a la izquierda del inicio de Rwave que supere el valor de  $Thf$ . Este punto se tiene que detectar dentro de un intervalo de 0.12 segundos, si no, se da por perdido.

### **7) Detección de la onda S**

Coincide con el primer mínimo local a la derecha del final de Rwave que supere el valor de  $Thf$ . Al igual que Qwave, este punto se tiene que detectar dentro de un intervalo de 0.12 segundos, si no, se da por perdido.



## 8) Detección del Onset y Offset de la onda P

La onda P, precede al complejo QRS y su inicio y fin se denominan onset y offset de P respectivamente.

Estos coinciden con los dos máximos locales consecutivos mayores que  $Thf$  a la izquierda de Qwave en la transformada (a la izquierda de Rwave en el caso de que Q no se haya detectado).

Se tiene que cumplir que haya un mínimo local entre el Onset y el Offset de P y que sea menor que  $-Thf$ . Este mínimo se denomina Ppeak.

## 9) Detección del Onset 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.

Estos coinciden con los dos máximos locales consecutivos mayores que  $Thf$  a la derecha de Swave en la transformada (a la derecha de Rwave en el caso de que S no se haya detectado).

Se tiene que cumplir que haya un mínimo local entre el Onset y el Offset de T y que sea menor que  $-Thf$ . Este mínimo se denomina Tpeak.

En la imagen 12.3 se puede ver un ejemplo de un complejo QRS, onda T y onda P señalado en la señal original sin ruido y su transformada morfológica, siguiendo los pasos del algoritmo.

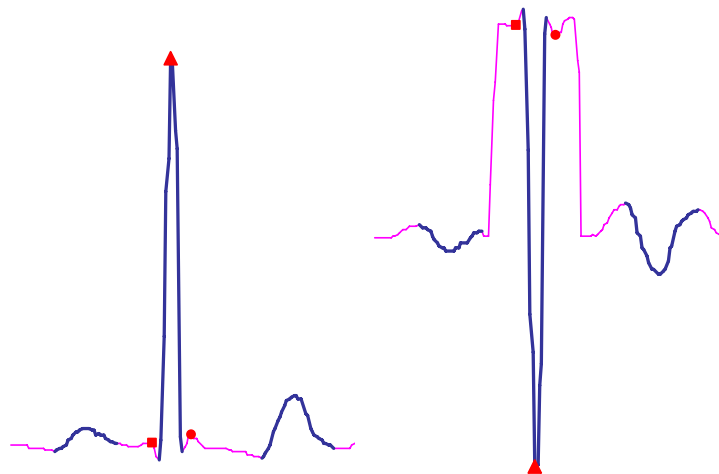


Fig. 12.3: Señal original y su transformada con las ondas características marcadas.

### 13. Adaptación del algoritmo al nodo

Como se explicó en el apartado 7, el nodo de 25 canales con el que vamos a probar el algoritmo propuesto, se trata de un dispositivo con recursos limitados y que aporta un dato nuevo cada milisegundo. Es necesario realizar una serie de modificaciones para adaptar el algoritmo propuesto por Yan [1], descrito en el apartado anterior, y conseguir que funcione correctamente.

La principal diferencia es que hay que modificar el algoritmo para que trabaje de forma dinámica. El programa en el que nos estamos basando analiza los datos (cargados previamente de un fichero obtenido de la base de datos de PhysioNet[4]) de forma estática, y aporta un conjunto de detecciones como solución.

En nuestro caso necesitamos un programa que trabaje de forma dinámica, recibiendo datos nuevos, aportados por el sensor a una frecuencia de 1000Hz, durante un periodo de tiempo indefinido. Por otro lado, el nodo solo cuenta con 2kB de memoria RAM, lo que limita la cantidad de datos almacenados a la vez y se hace imprescindible ir renovándolos de forma periódica. Debido a esto, se va a usar un buffer circular que se ira actualizando cada vez que reciba un nuevo dato y el algoritmo solo analizará esta parte de la señal. La longitud de este buffer ha sido elegida de forma que pueda contener al menos una detección completa (complejo QRS, ondas P y T) y parte del estudio de memoria se ha centrado en intentar reducir el tamaño de dicho buffer.

Lo primero a tener en cuenta es que hay que reducir la frecuencia de muestreo, ya que 1000 datos por segundo son excesivos e innecesarios. Además, el tamaño del buffer sería demasiado grande, con los consiguientes problemas de almacenamiento que esto supone. Por este motivo se reduce la frecuencia a 200Hz, tomando solo un dato de cada 5 proporcionados por el sensor.

A parte de la modificación anterior, se han incluido otros cambios para ajustar el programa a las limitaciones de memoria y velocidad de procesamiento del nodo.

- Transformación de los datos en punto flotante a enteros. El nodo consta de un microcontrolador muy sencillo (MSP430 [14]), que no tiene unidad en punto flotante, por lo que cualquier tipo de operación con decimales requiere una gran cantidad de soporte en software y además supone un gran coste computacional. Por este motivo, todos los datos y operaciones en punto flotante han sido transformadas a enteros con la consiguiente pérdida de precisión.

Aunque la señal de origen y la señal tras el filtro de ruido vienen representadas por valores enteros, el algoritmo tiene una división al realizar la transformada morfológica. La señal resultante de este paso ha sido reescalada por 10 para aumentar la precisión en un decimal. Tras varias pruebas, escalando la señal para otros valores que aumentaban la precisión en más decimales, se llegó a la conclusión de que basta con un decimal para conseguir una señal transformada válida para realizar las detecciones de las ondas características según el algoritmo, sin aumentar en exceso el número de bits necesarios para representar cada valor de la señal.

- Eliminación del paso 1. Nuestro propósito es probar la eficiencia y exactitud del algoritmo de detección de ondas características en un ECG y conseguir que se ejecute en el sensor, enviando paquetes con los resultados a la estación base, optimizando al máximo el uso de memoria que implica. El sensor devuelve señales con mucho ruido, y es necesario un filtro. Entonces, por un lado es necesario implementar un filtro específico para el nodo y optimizarlo y por otro crear un algoritmo que analice la señal. El objetivo de este trabajo es crear el algoritmo y por este motivo se desestima este paso.
- Eliminación del método Thresholding. Este método necesitaba usar algunos vectores de datos para realizar el histograma y ocupaba bastante memoria. Además, este método está pensado para una situación en la que se tiene la señal completa y por tanto todos los Rpeaks, mientras que al tratarse de un buffer circular, puede darse el caso de que no haya ningún Rpeak en el buffer de datos en ese momento y se elija un valor de Thr muy bajo. Tras numerosos estudios se vio que la mayoría de las señales tenían unos valores de umbrales  $Thf$  y  $Thr$  similares y se optó por fijarlos a  $Thr = 140$  y  $Thf = 5$ , para reducir el coste en memoria y tiempo de procesamiento que suponía este método.
- Por último, uno de los mayores cambios fue la división del método principal del algoritmo en partes, ya que el nodo no tenía suficiente velocidad de procesamiento para realizar un método tan largo en un solo ciclo. Ahora cada paso se realizará en un ciclo diferente (siendo un ciclo los 5ms que separan la recepción de cada dato del sensor).
- Elección del valor de la constante “s” para realizar la transformada para que se adapte a las señales del sensor. El nodo tiene una frecuencia de muestreo de 200Hz y s tiene que cumplir la siguiente condición:

$$W \text{ (Anchura de la onda característica)} * F_s \text{ (Frecuencia Muestreo)} > s$$

Con W entre 0.06 y 0.12. Aplicando este cálculo a nuestra frecuencia podemos elegir un valor de s entre 12 y 24. En ese rango, conviene elegir  $s = 16$ , que es potencia de 2, para evitar tener una división y poderla sustituir por un desplazamiento que tiene menor coste computacional.

Fórmula inicial:

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

Fórmula final:

$$M_f^{ds}(x) = \max\{f(t) \mid t \in [x-s, x+s]\} + \min\{f(t) \mid t \in [x-s, x+s]\} - 2f(x) \gg 4$$

El algoritmo en sí mismo era insuficiente para conseguir nuestro propósito: una herramienta de autodiagnóstico. Por este motivo, se añadió un último paso al algoritmo, en el que una vez obtenida una detección, se aplican las reglas de Normalidad explicadas en el apartado 11 para comprobar si el complejo QRS incumplía alguna de ellas.

Por otro lado, en el algoritmo de Yan [1] no se especifica la forma de expresar las detecciones. Para nuestra modificación, es interesante tener una referencia temporal. En las primeras versiones, los resultados venían expresados en posiciones con respecto al buffer de datos almacenado en el nodo. Pero en versiones posteriores, se incluyó un reloj, que se incrementa en función de los datos de entrada (1 segundo cada 200 datos), y las detecciones se expresan en función de este reloj. El Rpeak, que es la primera onda detectada se toma como referencia, expresándose en horas, minutos, segundos y milisegundos, mientras que el resto de ondas se expresan en función de la distancia, en milisegundos, que las separa del Rpeak.

Result	H	Min	Seg	Milisg	Rw0	Rw1	Q	S	onsetP	offsetP	onsetT	offsetT
			RA			PA			TA			

Fig. 13.1: Señal original y su transformada con las ondas características marcadas.

El formato de la detección se muestra en la imagen 13.1, donde:

- **Result:** Resultado que depende de la validación de la detección y las Reglas de Normalidad. En la *tabla 13.1* se pueden ver los diferentes valores que puede tomar.
- **H:** hora en la que hay un Rpeak.
- **Min:** minuto en la que se encuentra el Rpeak dentro de esa hora “Ho”
- **Seg:** 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, PA, TA:** Amplitud de las ondas R, P y T respectivamente (valor de Rpeak 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 13.1. Valores de la variable Result

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

Para probar las señales de esta base de datos que están muestreadas a una frecuencia de 250Hz, se ha tenido que transformar las señales para que tengan las mismas características que las que nos proporcionaría el sensor del nodo, que muestrea las señales a 1000Hz. Solo se toman un dato de cada 5, por lo que se necesita reducir la frecuencia de muestreo de las señales tomadas de la base de datos a 200Hz.

Para realizar esta reducción, se utiliza el método *Resampling* al que le se le indica por parámetro la frecuencia de entrada y de salida. Este método elimina un valor de cada 5, y a continuación recalcula los valores de la señal para que sea uniforme.

El programa cargado en el nodo se ha comparado con el algoritmo estático MMD [1] original, un algoritmo basado en umbrales adaptativos (TD [18]) y un algoritmo basado en la transformada de wavelet (WD [19]). Para comprobar la exactitud de las detecciones se han empleado tres parámetros:

- El error medio (mean),
- La desviación estándar ( $\sigma$ ),
- Y 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.

Algoritmo	Parámetros	Pon	Poff	QRSon	QRSoff	Ton	Toff
Adaptación	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
	$\sigma$ (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
	$\sigma$ (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
	$\sigma$ (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
	$\sigma$ (ms)	12.7	11.9	7.7	8.7	13.5	18.8
CSE	$\sigma$ (ms)	10.2	10.7	6.5	11.6	-	30.6

Tabla 13.2. Comparación de los resultados obtenidos

En la tabla 13.2 se comparan los valores obtenidos para cada uno de los algoritmos y la desviación estándar tolerable dada por el comité de Common Standards for Electrocardiography (CSE).

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 resultados no se alejan demasiado de los obtenidos en el algoritmo propuesto por Yan Sun [1].

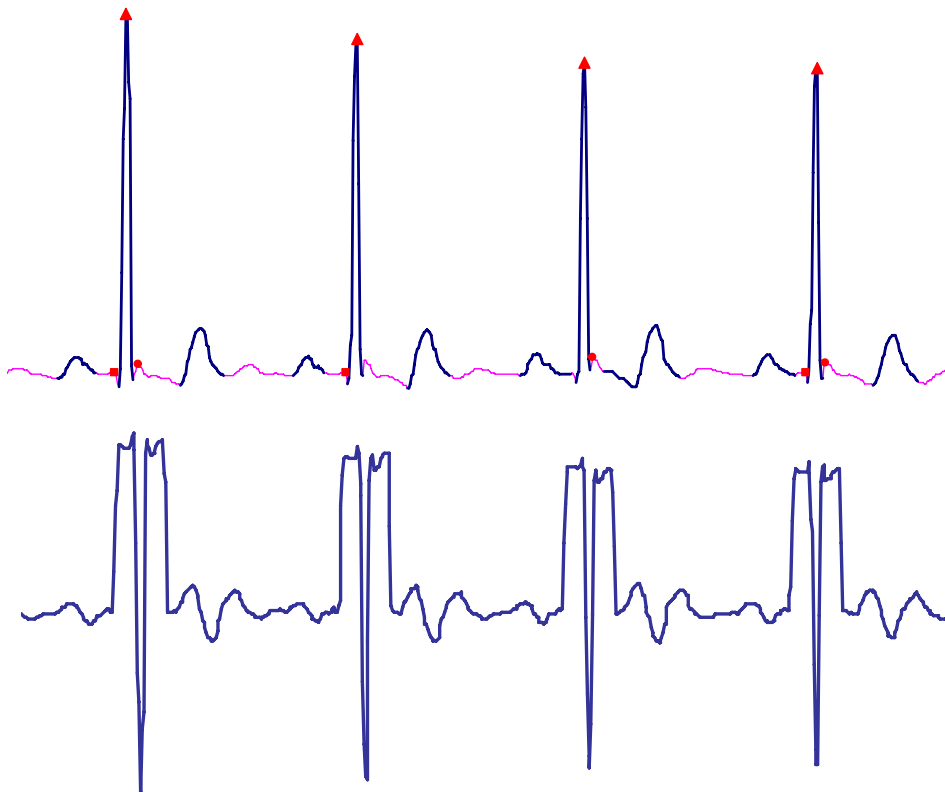


Fig. 13.2: ECG de la base de datos QT [4] con las detecciones y su transformada.

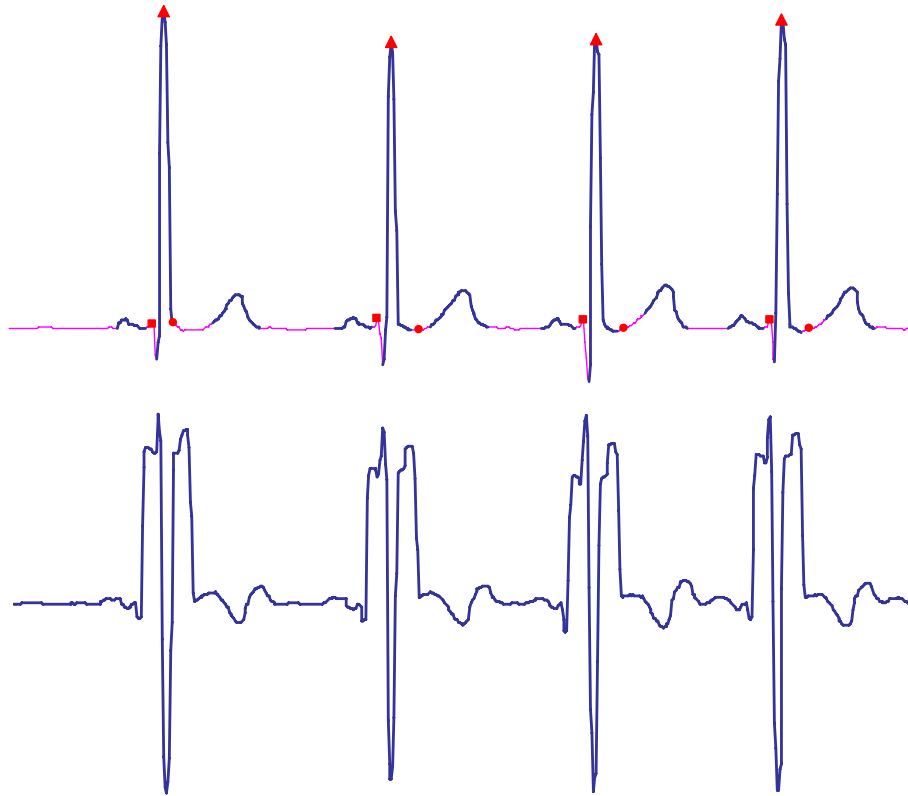


Fig. 13.3: ECG de la base de datos QT [4] con las detecciones y su transformada

En las imágenes 13.2 y 13.3 se muestran dos señales analizadas con el algoritmo implementado, su transformada morfológica asociada, y las detecciones de las ondas características obtenidas.

## **14. Mejoras en Memoria.**

Uno de los principales problemas del nodo es la limitación de memoria. Tiene que ser pequeño para que sea ligero, discreto y cómodo de llevar, por este motivo todos sus componentes, incluida la memoria tienen que tener un tamaño reducido. Por este motivo, se hace necesario estudiar cuales son los mayores problemas, derivados del uso de memoria y la cantidad de procesamiento, que pueden aparecer a la hora de diseñar aplicaciones que analicen la señal ECG.

Las primeras versiones fueron implementadas sin tener en cuenta el consumo o la capacidad de procesamiento del nodo, y hubo que retocarlas mucho hasta que se consiguió una primera versión que no necesitara más de 2kB de memoria dinámica. Con esto se demuestra su increíble limitación, ya que un programa normal no es una aplicación válida para el nodo, y hay que retocarlo para adaptarlo a sus limitaciones.

Al estar trabajando con buffers que almacenan la señal, esto se hace muy complicado, y el estudio de memoria se ha centrado en reducir el número de buffers y su tamaño.

### **14.1. Versión 1.**

La primera versión del programa fue programada sin tener en cuenta las limitaciones del nodo para ver los resultados que tenía analizar las señales de PhysioNet [4] con el algoritmo elegido. Se trata de una versión en la que no se desprecia ninguno de los pasos, y se usan tantos buffers como se necesiten, sin reusar ninguno de ellos.

La longitud de los buffers para esta versión es de 1024 posiciones de 16 bits ( $1024 * 2 \text{ bytes} = 2 \text{ kB}$  cada buffer, la memoria RAM queda desbordada solo con uno de ellos). La longitud fue tomada al azar, sin tener en cuenta, como he dicho, las limitaciones de memoria.

Se usan 3 buffers principales: uno para la señal de origen, otro para la señal sin ruido y otro para la transformada. En un principio se vio necesario guardar los tres debido a que el filtro de ruido y la transformada utilizan una ventana de datos para calcular cada posición. Por otro lado, para las operaciones del filtro de ruido y de thresholding era necesario mantener otros 2 buffers auxiliares. Con todo esto ya estamos hablando de 10kB de memoria.

Aunque no son tan significativos, también hay que tener en cuenta los 4 elementos estructurales Bo, Bc, B1 y B2 de 40, 60, 5 y 5 posiciones de 8bits respectivamente, y el buffer de salida con las detecciones de 16 posiciones de 16 bits. Todo en total suma más de 10kB de memoria, muy lejos de los 2kB disponibles y que se hace imprescindible reducir.

### **14.2. Versión 2.**

Esta versión se mejora considerablemente la memoria utilizada, sencillamente disminuyendo la longitud de los buffers de 1024 a 300 datos. Se eligió este valor para asegurar que en un buffer entra una detección completa (QRS, ondas P y T), requisito



fundamental para que funcione el algoritmo. Teniendo en cuenta que la máxima longitud de este complejo en las ondas probadas es de 0.82 segundos, a una frecuencia de 200Hz, nos sobrarían 136 posiciones. Estas se reservan como márgenes de datos inválidos (68 posiciones) al comienzo y al final del buffer que contiene la señal procesada, y que se pierden al usar ventanas para hacer los cálculos del filtro y de la transformada.

Por otro lado se reduce el número de buffers, tras concluir que el vector que almacena la señal transformada no es necesario. Éste se utilizaba en un principio para almacenar la señal procesada y no tener que recalcularla cada vez que se introducía un nuevo dato, y así reducir la cantidad de procesamiento. Sin embargo, sabiendo que la limitación de memoria es mayor y que en dicho método solo se analizan  $2*s$  datos (con  $s=16$ , ver apartado 13), es preferible calcular la transformada de cada dato cuando se necesite en vez de almacenarla.

Haciendo el cálculo para esta reducción, la nueva versión necesitaría  $4*300*2$  bytes = 2.34kB de memoria para almacenar los buffers. Se ha reducido significativamente con respecto a la versión 1, sin embargo aún se aleja bastante de ser válido para el nodo porque también habría que tener en cuenta los elementos estructurales y el resto de variables que usa el programa.

### 14.3. Versión 3.

Visto que las mayores mejoras se obtienen reduciendo el tamaño de los buffers, para esta versión se analiza el desperdicio de bits por parte de los datos almacenados en ellos, para crear buffers comprimidos.

Los buffers están formados por posiciones de 16 bits, pero es imposible reducirlos a posiciones de 8 bits ya que son insuficientes para representar todos los valores (0 a 255 ó -128 a 127). Sin embargo, analizando los valores máximos y mínimos de la señal y sus transformadas se comprobó que con 12bits era suficiente para representarlos (0 a 4095 ó -2048 a 2047). Visto esto, se llega a la conclusión de que 12 bits se pueden representar con un dato de 8 bits y la mitad de otro. De esta forma, cada 3 valores de 8 bits se pueden representar 2 valores de 12bits.

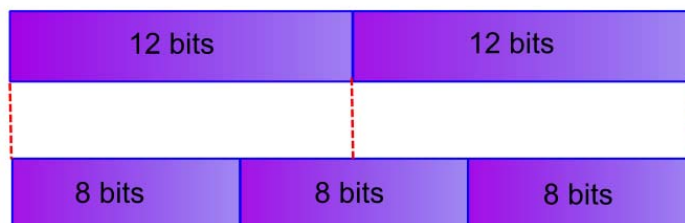


Fig. 10.1: Compresión de 2 valores de 16bits a 3 de 8, despreciando los 4 más significativos.

Llegado a este punto se puede comprimir un buffer de 300 posiciones de 16 bits en uno de 450 posiciones de 8 bits ( $300*3/2$ ). Esto reduce la memoria que ocupa un buffer de 600 a 450bytes, y en total se pasa de 2.34kB a 1.75kB.

A pesar de la compresión de los buffers, la reducción de la memoria sigue siendo insuficiente si tenemos en cuenta el resto de variables, pero esta medida es bastante importante porque aunque el ahorro de espacio no es tan significativo como en el caso anterior, si que aprovecha mejor el espacio de memoria del nodo, evitando tener bits inutilizados.

Sin embargo, una desventaja de esta es que se necesita descomprimir y comprimir los datos cada vez que se accede a los buffers y esto exige mayor cantidad de cómputo cada vez que se pasa el algoritmo.

#### 14.4. Versión 4.

El filtro de ruido es el que más cómputo y más buffers utiliza, por lo que en esta versión se suprime y se carga el programa con señales ya filtradas. Esta medida es temporal, a la espera de diseñar un filtro más sencillo y lo suficientemente potente como para limpiar la muestra recogida por el sensor. Mientras, nos sirve para hacer las pruebas y se consigue un a primera versión portable.

Ahora solo tenemos dos buffers, el que guarda la señal original (ya filtrada) y un auxiliar para el thresholding, que ocupan 900bytes. Esto, sumando a todas las variables y el código, según el compilador para el protocolo MAC dinámico y estático da los siguientes resultados en bytes:

	ROM (bytes)		RAM (bytes)	
	Dinámico	Estático	Dinámico	Estático
<b>Versión 4</b>	13036	12412	1405	1323

Tabla 14.1: uso de memoria de la versión 4 del algoritmo MMD dinámico para el protocolo MAC dinámico y estático.

#### 14.5. Versión 5.

La versión anterior es portable pero no funciona debido a que el procesador del nodo no tiene suficiente velocidad de procesamiento para completar todo el algoritmo a tiempo. Esto es debido a que el algoritmo se aplica sobre los datos almacenados cada vez que se recibe un nuevo dato del sensor. Esto se hace mediante un timer (evento) que salta cada 1ms, y que cada 5 veces hace una llamada al algoritmo para que introduzca un nuevo dato y complete el análisis. Este tiempo, de tan solo 5ms, es insuficiente para completar el algoritmo y se solapan unas llamadas con las anteriores.

Debido a esto, se decidió dividir el algoritmo en 8 pasos, de forma que cada fase del algoritmo se hace en un ciclo diferente:

- Paso 1: Introduce un nuevo dato en el buffer circular y actualiza los contadores y punteros. Calcula el thresholding en caso de que fuera necesario.

- Paso 2: Detección del Rpeak, si no lo encuentra salta de nuevo al paso 1. Si lleva varias veces sin encontrarlo, da error y pasa al paso 8.
- Paso 3: Detección de Rwave. Si no lo encuentra salta al paso 1.
- Paso 4: Detección de Qwave.
- Paso 5: Detección de Swave.
- Paso 6: Detección de Pwave.
- Paso 7: Detección de Twave. Si no la encuentra, deja tiempo para que entren más valores en el buffer mientras Pwave no se salga de este, sino dará error en el paso 8.
- Paso 8: Validación de la detección y envío de resultados a la estación base.

Los resultados obtenidos tras esta división son:

	ROM (bytes)		RAM (bytes)	
	Dinámico	Estático	Dinámico	Estático
<b>Versión 5</b>	15774	15150	1439	1357

Tabla 14.2: uso de memoria de la versión 5 del algoritmo MMD dinámico para el protocolo MAC dinámico y estático.

Son peores que los de la versión anterior, pero hay que tener en cuenta que se han tenido que añadir más variables y ahora el código es más largo. Por otro lado, esta versión ya es portable y funciona en el nodo.

Además, se han añadido otras mejoras como la incorporación de un reloj interno para calcular el tiempo real de las detecciones en horas, minutos, segundos y milisegundos. Este reloj incrementa un segundo cada 200 datos, y los cálculos están hechos en función de los contadores del buffer de la señal (apartado 13).

## 14.6. Versión 6.

La versión anterior presenta un problema: como solo se cargan datos en el primer paso, en los 7 siguientes se pierden los datos que aporta el sensor. Además el procesamiento es muchísimo más lento, ya que tarda 8 ciclos en realizar lo que antes solo realizaba en 1 (cuando se habla de ciclos se refiere a 5ms, es decir, las 5 activaciones del timer). En la versión 6 esto se soluciona guardando los datos recibidos en un buffer de 8 posiciones y cargándolos en el buffer circular en el paso 1. No se pueden cargar paso a paso ya que esto variaría todos los contadores y la detección entonces ya no sería correcta. De la otra forma no hay problema, incluso es mejor porque se aportan 8 datos nuevos a cada análisis y el algoritmo es incluso más rápido.

Tras probar con varias señales, se notó que el método que realiza el thresholding consume excesiva memoria en vano, ya que la mayoría de las muestras tienen umbrales similares y estos valores se pueden establecer como constantes. Además, este método empezó a dar muchos problemas a raíz de reducir el tamaño del buffer de la primera versión de 1024 datos a la segunda de 300, ya que al tener tan pocos datos, en algunas ocasiones no hay ningún Rpeak en el buffer, y toma valores de Thr muy bajos y el algoritmo detecta falsos picos R.

Como se ve, el hecho de eliminar dicho método reduce significativamente el uso de memoria RAM.

	ROM (bytes)		RAM (bytes)	
	Dinámico	Estático	Dinámico	Estático
<b>Versión 6</b>	15868	15244	1003	921

Tabla 14.3: uso de memoria de la versión 6 del algoritmo MMD dinámico para el protocolo MAC dinámico y estático.

### 14.7. Versión 7.

Esta vez se aplica una de las mejoras más importantes que consigue reducir el buffer a más de una séptima parte de su versión anterior. El algoritmo consiste básicamente en buscar una serie de máximos y mínimos que superan unos umbrales (Thr para el RPeak y que es mayor que Thf para el resto de ondas). La idea entonces es almacenar únicamente dichos puntos, y desechar el resto de valores de la señal que realmente no son necesarios. Esta medida, no solo disminuye enormemente el uso de memoria, sino que también acelera el procesamiento de la señal ya que, ahora no tiene que recorrer un buffer de 300 datos cada vez que quiere detectar un punto.

Para llevarlo a cabo se necesitan varias modificaciones. Primero se reduce el buffer original, que sigue siendo necesario para guardar la ventana necesaria para hacer la transformada de los datos. Se necesita un buffer de 40 posiciones, para calcular las detecciones en horas, minutos, segundo y milisegundos, teniendo en cuenta que cada vez que se llenan 5 buffers es un segundo. Este buffer esta comprimido, es decir, son 40 datos que se guardan en un buffer de  $40 * 3/2 = 60$  posiciones.

Hasta ahora las posiciones de cada dato se calculaban en función de su posición del buffer de 300 elementos y el reloj del sistema. Ahora la complicación es que el buffer tiene menos posiciones, pero por otro lado es más sencillo conservar un buffer virtual de 200 posiciones, es decir, calcular todas las posiciones de los datos en modulo 200 en vez de 40.

Por otro lado se crean cuatro vectores de 24 posiciones, tres de ellos con datos de 16bits y uno con datos de 8, para guardar la información de los máximos y los mínimos.

- maxMin[]: guarda las posiciones en modulo 200.
- valuesfp[]: guarda el valor de dicha posición en la señal original para devolver las amplitudes del Rpeak, las ondas P y T.
- values[]: guarda el valor de la transformada para comparar con los umbrales en los métodos de detección.
- isMax[]: se necesita este buffer auxiliar para diferenciar entre máximos y mínimos. (toma valor 1 si es máximo).

La memoria necesaria para esta versión disminuye en comparación con la anterior.

	ROM (bytes)		RAM (bytes)	
	Dinámico	Estático	Dinámico	Estático
<b>Versión 6</b>	14690	14066	789	709

Tabla 14.4: uso de memoria de la versión 7 del algoritmo MMD dinámico para el protocolo MAC dinámico y estático.

### 14.8. Versión 8.

Esta es la versión definitiva, para la que se han obtenido mejores resultados. Se reduce al máximo la longitud de los vectores que guardan los máximos y mínimos a 18 elementos.

A la hora de reducir el tamaño de este vector, hay que tener en cuenta que al menos debe contener una pulsación completa. Para esto necesita guardar como mínimo 11 posiciones (3 para la onda P, 3 para la onda T, 3 para el QRS, y dos para la onda R).

Sin embargo este valor es demasiado restrictivo, ya que en ocasiones hay más de un mínimo o máximo separando dos de los puntos característicos, pero que se desestiman porque no es lo que se está buscando en ese momento en el algoritmo. Por ejemplo, entre el onset y el Offset siempre hay al menos un mínimo, pero puede suceder que haya más de uno, que no se tienen en cuenta. Esta situación se puede dar entre Q y la onda P y entre S y la onda T (un máximo y un mínimo por cada una) y entre el onset y el offset de las ondas P y T (puede haber un mínimo adicional, si no, se sale de los tiempos normales para encontrar una onda P o T y corre el peligro de confundirla con las ondas pertenecientes al pulso anterior o siguiente). En total son entonces 6 posiciones adicionales a las 11 mínimas.

	ROM (bytes)		RAM (bytes)	
	Dinámico	Estático	Dinámico	Estático
<b>Versión 6</b>	15206	14582	759	679

Tabla 14.5: uso de memoria de la versión 8 del algoritmo MMD dinámico para el protocolo MAC dinámico y estático.

## 15. Estudio de los resultados

Una vez obtenidos los resultados para las ocho versiones desarrolladas, se puede hacer una comparativa para comprobar la mejora de las versiones finales con respecto a las iniciales, en concepto de uso de memoria.

Vemos que la pérdida de precisión en la mayoría de los casos no ha alterado las detecciones y ha proporcionado un importante ahorro de la memoria utilizada. Se ha conseguido reducir la memoria RAM a más de la mitad de la que se usaba en las primeras versiones. Y ha sido posible que el algoritmo funcionara en el nodo a pesar de todas las modificaciones sufridas

Aunque aun hay problemas de memoria para poder usar el filtro, se podría usar otro método para filtrar la señal, menos óptimo que no conllevara tanto procesamiento y uso de memoria. Se están desarrollando nuevos nodos con un aumento de las características de memoria (10 kB de RAM) que permitirán usar el filtro.

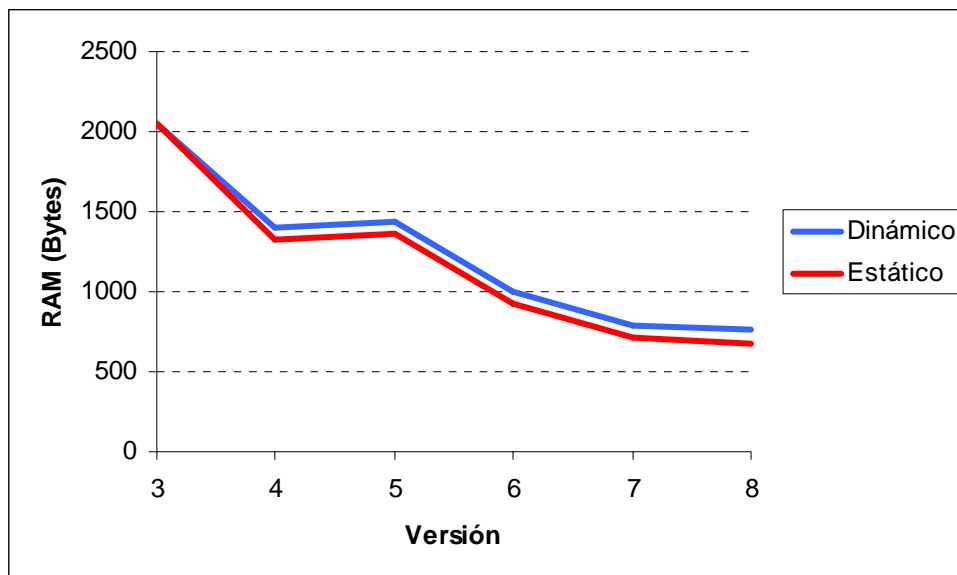


Fig. 15.1: Uso de RAM en el nodo de las versiones 3 a la 8 del algoritmo para ambos protocolos MAC dinámico y estático.

Hecho el análisis se pueden concretar cuales son las modificaciones que tienen más impacto y suponen más ahorro en memoria y carga computacional que es necesario aplicar para conseguir un programa portable en el nodo que estamos utilizando.

- Reducción al mínimo del **número de datos** necesarios en los buffers. En el gráfico 15.2, se observa la notable reducción desde la primera versión a la última. En esta se ha tenido en cuenta el tamaño del buffer de datos (40) más la suma de los buffers necesarios para realizar esta reducción y como se explicó en el punto 10.8.

Se puede concluir entonces, que es importante almacenar únicamente los datos necesarios aunque en principio el algoritmo esté planteado de otra forma. En este caso, solo se necesitan los máximos y mínimos locales, y por este motivo, es posible llegar a la versión 7 (en la que se tienen en cuenta tanto el buffer de datos, como los 4 buffers de máximos y mínimos).

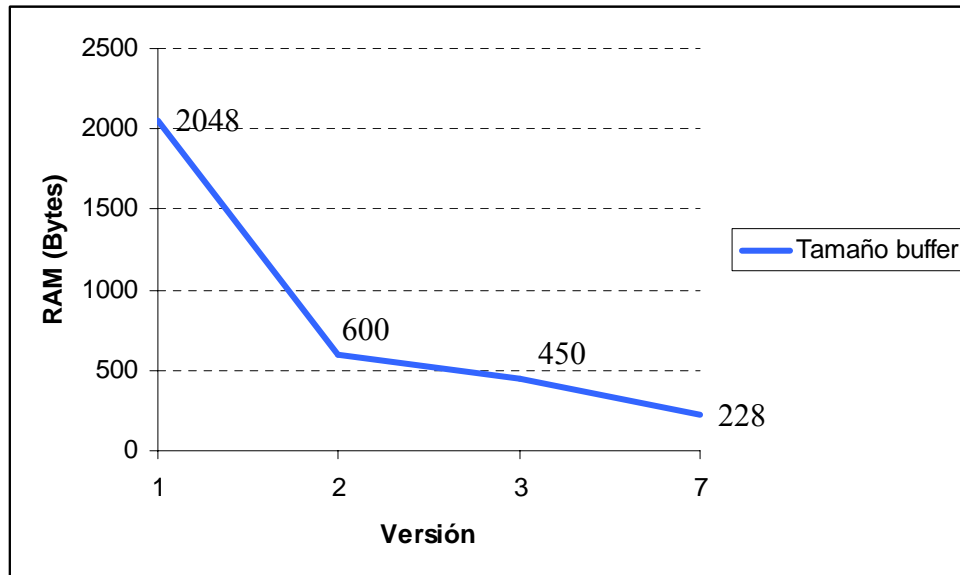


Fig. 15.2: Tamaño de los buffers usados para las diferentes versiones.

- Aprovechamiento máximo del área de memoria mediante las técnicas de **compresión y descompresión de datos** explicadas en el apartado 10.3. En el gráfico 15.2 se observa la reducción del número de bytes por buffer que supone el uso de esta técnica. Aunque en este caso sólo se ha aplicado a los vectores que almacenan la señal, se podría aplicar en otras variables (por ejemplo a los vectores auxiliares de la versión 7) para reducir todo lo posible el uso de memoria en caso de que fuera necesario.
- **Reutilización** de buffers y variables. Esto es importante cuando la memoria es crítica, para reducir el número de variables necesarias.
- **División del algoritmo** en pasos en caso de que sea posible: esto es importante debido a que el nodo tiene un procesador limitado y que solo trabaja a 8MHz, ya que al tener un sistema basado en eventos (TinyOS [5]), no le da tiempo a completar todas las operaciones para algoritmos muy largos y se acumula el trabajo.
- Sustitución de las **divisiones por desplazamientos**: esto es una técnica general hardware, pero que toma especial importancia en el nodo para reducir el tiempo de ejecución de cada método.

## 16. Conclusiones

Las redes corporales de sensores inalámbricos son una buena alternativa para el seguimiento de personas con problemas de salud. Hasta ahora se optaba por enviar todos los datos recogidos, y era la estación base la que los analizaba. Esto no es un buen enfoque ya que las transmisiones consumen demasiada energía y se reduce el tiempo de vida de los nodos. Por este motivo, es interesante incluir aplicaciones que analicen los datos y reduzcan el número de transmisiones.

Las limitaciones de memoria y capacidad de procesamiento son uno de los mayores problemas de las redes de sensores inalámbricos, y dificultan la creación de aplicaciones capaces de ejecutarse en los nodos.

En este proyecto se ha conseguido integrar en uno de los nodos que compone la red, un algoritmo que realiza un análisis en tiempo real de los datos leídos por los sensores y reduce notablemente el número de transmisiones necesarias entre el nodo y la estación base.

Con este fin, se ha utilizado un algoritmo de análisis de ECG, diseñado para trabajar con electrocardiogramas completos, y se ha realizado una serie de modificaciones y optimizaciones en memoria, y como resultado se ha conseguido implementar un algoritmo dinámico que analiza los datos online y que cumple con las restricciones de memoria del nodo.

Por último se ha añadido funcionalidad al algoritmo, de forma que es capaz de detectar una serie de problemas cardíacos comunes, y envía solo la información necesaria de estos a la estación base.

La aplicación resultante es una herramienta de autodiagnóstico lo suficientemente potente como para detectar las ondas características más importantes del ECG e informar sobre los posibles problemas cardíacos del paciente, analizando estas detecciones.



## **Apéndices.**



## Apéndice A: Instalación de TinyOS y MSP430-GCC en Windows.

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

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 (c:\) para evitar tener rutas excesivamente largas o con espacios que dificultan la instalación.

1. Instalar Tinyos-1.1.0-1is.exe. Este ejecutable incluye Cygwin, Java y TinyOS 1.1.0. Esto tarda algún tiempo.
2. Abre CygWin. Este programa es un entorno desarrollado por Cygnus Solutions para proporcionar un comportamiento similar a los sistemas Unix en Windows. A continuación se instalarán el resto de componentes.

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

Esta ruta corresponde a la carpeta donde tenemos guardados todos los archivos requeridos para la instalación y debe ser ajustada según proceda.

3. Mejorar 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 error `about_getreent not found in cygwin1.dll`.
5. Reiniciar CygWin, así evitaras posibles fallos.
6. Actualizar TinyOS a la versión 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 necesitadas por 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 protocolo MAC: Incluir *NetworkGenericComm* dentro de la carpeta *C:\tinyos\cygwin\opt\tinyos-1.x\tos\lib\*.

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* e *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
```



## **Apéndice B: TOSSIM.**

TOSSIM [15] es el simulador de TinyOS. Éste permite a los usuarios probar y analizar los algoritmos en un medio controlado antes de portar las aplicaciones al nodo, y permite usar herramientas de debug.

El principal propósito de TOSSIM es proveer a las aplicaciones de TinyOS de una herramienta de simulación de gran fidelidad. Por lo que se centra en la simulación de TinyOS y su modelo de ejecución, más que en simular el mundo real. Sin embargo, como otras herramientas de simulación, no es totalmente fiable, ya que asume muchas cosas y simplifica algunos comportamientos.

Principales características:

- **Fidelidad:** Por defecto, TOSSIM captura el comportamiento de TinyOS a un nivel muy bajo. Simula la red a nivel de bit y simula cada captura ADC individual y cada interrupción del sistema.
- **Tiempo:** Mientras TOSSIM permite ejecuciones de tiempo con precisión, no simula el tiempo de ejecución, es decir, el código se ejecuta instantáneamente.
- **Modelos:** TOSSIM provee de abstracciones de ciertos fenómenos del mundo real (tales como un bit error). Con algunas herramientas fuera de la simulación, se pueden manipular estas abstracciones para implementar el modelo que se quiera usar. De esta forma TOSSIM permanece flexible a las necesidades de muchos usuarios, y además consigue una simulación simple y eficiente.
  - **Radio:** TOSSIM no simula la propagación de la radio, en vez de eso, provee una abstracción del modelo de radio y lo mapea a estos errores de bit. Teniendo tasas de error directas se pueden modelar conexiones asimétricas fácilmente. Los errores de bit independientes causan que los paquetes más largos tengan una mayor probabilidad de corrupción, y la probabilidad de pérdida de paquete es independiente.
  - **Consumo:** TOSSIM no modela el consumo de energía, sin embargo es muy simple añadir componentes que provean información cuando los estados de consumo cambien. Tras una ejecución, se puede aplicar un modelo de consumo a estas transiciones, calculando el consumo general. Sin embargo, como TOSSIM no incluye el tiempo de ejecución de la CPU, no se puede calcular fácilmente el consumo de ésta.
- **Compilación:** TOSSIM compila directamente el código de TinyOS. Para simular un protocolo o un sistema, se debe escribir una implementación de TinyOS.
- **Imperfecciones:** TOSSIM realiza varias simplificaciones del comportamiento de TinyOS. Esto significa que un código que ejecuta en una simulación, puede que no funcione en un nodo real.
- **Networking:** TOSSIM simula la pila de red RFM mica de 40Kbits.

- **Confianza:** Los resultados obtenidos no deben ser considerados fidedignos ya que las redes de TinyOS tiene un comportamiento muy complejo y variable.

En resumen, TOSSIM no debe ser considerado como una herramienta de simulación definitiva, pero nos sirve para hacernos una idea de cómo funcionan los algoritmos antes de probarlos en los nodos.

## 1. Compilando y Ejecutando una Simulación.

Para compilar una aplicación hay que situarse en el directorio donde esta guardada y hacer make

```
bash$ cd $TOSROOT/apps/carpetaPrograma
bash$ make pc
```

De esta forma solo compilas una simulación de la aplicación. El ejecutable se llama main.exe y esta en la carpeta */build/pc*. Tras esto puedes ejecutar con una serie de opciones.

```
bash$ ./build/pc/main.exe [options] num_nodes > [out.txt]
```

Donde options puede ser:

-h, --help	
-gui	Pausa la simulación esperando a GUI que se conecte
-a=<model>	Especifica el modelo ADC: generis (por defecto) o random
-b=<sec>	Los nodos se inician en los primeros <i>sec</i> segundos (10 por defecto)
-ef=<file>	Usa el fichero <i>file</i> para eeprom, si no usa uno anónimo
-l=<scale>	Ejecuta la simulación a <i>scale</i> veces del tiempo real
-r=<model>	Especifica el modelo de radio: simple (default), static o lossy
-rf=<file>	Especifica el fichero de entrada para el modelo lossy (lossy.nss)
-s=<num>	Solo inicia <i>num</i> nodos
-t=<sec>	Ejecuta la simulación durante <i>sec</i> segundos virtuales
num_nodes	Número de nodos a simular
out.txt	Fichero donde guardar la salida por pantalla de la simulación

### 1.1. Debug.

Tossim consta de una herramienta de debug en tiempo de ejecución. Cuando se inicia una simulación, se lee de la variable de entorno DBG para determinar los modos de debug activados. Esta variable tiene que ver con la depuración, dependiendo del valor que tenga se nos mostrarán ciertos mensajes de información cuando simulamos una aplicación. Los valores que puede tomar DBG son los siguientes:

```
all: Enable all available messages
boot: Simulation boot and StdControl
clock: The hardware clock
task: Task enqueueing/dequeueing/running
```



sched: The TinyOS scheduler  
sensor: Sensor readings  
led: Mote leds  
crypto: Cryptographic operations (e.g., TinySec)  
route: Routing systems  
am: Active messages transmission/reception  
crc: CRC checks on active messages  
packet: Packet-level transmission/reception  
encode: Packet encoding/decoding  
radio: Low-level radio operations: bits and bytes  
logger: Non-volatile storage  
adc: The ADC  
i2c: The I2C bus  
uart: The UART (serial port)  
prog: Network reprogramming  
sounder: The sounder on the mica sensor board  
time: Timers  
sim: TOSSIM internals  
queue: TOSSIM event queue  
simradio: TOSSIM radio models  
hardware: TOSSIM hardware abstractions  
simmem: TOSSIM memory allocation/deallocation (for finding leaks)  
usr1: User output mode 1  
usr2: User output mode 2  
usr3: User output mode 3  
temp: For temporary use

Por defecto el valor de DBG es “all”, si no la modificamos se nos van a mostrar todos los mensajes posibles en la simulación, con lo cual la misma va a tardar muchísimo tiempo. Para modificar el valor de DBG ejecutamos, por ejemplo, el siguiente comando:

```
bash$ export DBG = boot,led
```

Al ejecutar este comando estamos diciéndole al simulador que durante la ejecución de la aplicación queremos que nos muestre por pantalla sólo los mensajes que tienen que ver con el arranque del nodo y con los leds.

Cuando programamos una aplicación podemos poner en cualquier sitio de nuestro código sentencias del tipo:

```
dbg(DBG_USR1, "Entro por aquí");
```

Estas sentencias son las que se imprimen por pantalla cuando ejecutamos nuestra aplicación, y funcionan de forma similar a la función *print* de C. Los tipos *usr1*, *usr2* y *usr3* se reservan a mensajes que el usuario quiera agregar en su programa para luego depurarlo, es decir, si añadimos sentencias en nuestro código como la anterior y luego le damos a DBG el valor “usr1”, vamos a ver sólo los mensajes que hemos puesto nosotros con el tipo *usr1*.

## 1.2. Monitorización de la red e Inyección de paquetes.

Para interactuar con una red simulada, debes usar el *SerialForwarder*, la herramienta estándar de TinyOS. Para trabajar con TOSSIM, la fuente de entrada del *SerialForwarder* se debe configurar apropiadamente. Hay dos modos:

- Comunicación a través de un puerto en serie con el nodo 0: Este modo interacciona con el nodo 0 a través de su puerto en serie. Los programas conectados al *SerialForwarder* pueden leer los mensajes que este nodo envía a su puerto serie y los mensajes que llegan a través de dicho puerto. La forma de activarlo es: "tossim-serial" en el campo "Mote Communications" o "-comm tossim-serial" en la línea de comandos.
- Network snooping: Los programas conectados al *SerialForwarder* pueden leer todos los mensajes de radio enviados en la red, y pueden enviar mensajes a cualquier nodo. Para activarlo: "tossim-radio" en el campo "Mote Communications" o "-comm tossim-radio" por línea de comandos.

## 2. Modelos de Radio

TOSSIM simula la red a nivel de bit. Inicialmente usaba las implementaciones de las componentes de TinyOS casi idénticas a la mica 40Kbit RFM-based, en la actualidad ya se simula el comportamiento de la mica2 CC1000-based. En TOSSIM una señal de red es 1 ó 0, y las colisiones se modelan como una o-lógica. Esto significa que la distancia no afecta a la fuerza de la señal.

Hay dos modelos de radio:

- Simple: coloca todos los nodos en una única celda. Cada bit transmitido se recibe sin error. Sin embargo, dos nodos pueden transmitir al mismo tiempo, y las dos señales se solaparán, pero esto es muy poco probable debido al protocolo CSMA de TinyOS.

Este modo es útil para probar algoritmos single-hop y componentes de TinyOS para corregirlos.

- Lossy: Este modelo coloca los nodos en un grafo dirigido, que puede ser especificado en el archivo *lossy.nss* u otro archivo especificado mediante el flag *-rf*. Cada arista tiene un valor de 0 a 1 que representa la probabilidad de que un bit enviado esté corrupto. El fichero tiene el siguiente formato:

```
<mote ID>:<mote ID>:bit error rate
<mote ID>:<mote ID>:bit error rate
<mote ID>:<mote ID>:bit error rate ...
```

El modelo de radio de la plataforma Mica2 está implementado en el fichero: *tinys1.x\tos\platform\pc\CC1000Radio\rfm\_model.c*.

Ahí se lee el fichero (por defecto, llamado "*lossy.nss*") con las tasas de pérdida 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: *tinys1.x\tos\platform\pc\nRF2401RadioShockBurst*

## 2.1. Errores de bit y de paquete.

La fórmula para calcular las tasas de errores de paquete ( $E_p$ ) a partir de la tasa de errores de bit ( $E_b$ ) para la mica RFM 40Kb snack es la siguiente:

$$\begin{aligned} E_p &= 1 - (S_s * (S_e)^d) \\ S_s &= (1 - E_b)^9 \\ S_e &= (1 - E_b)^8 + (8 \cdot E_b * (1 - E_b)^{12}) \\ E_p &= 1 - ((1 - E_b)^9 * ((1 - E_b)^8 + (8 * E_b * (1 - E_b)^{12}))^d) \end{aligned}$$

Donde:  $S_s$  es la probabilidad de éxito del símbolo inicial,  
 $S_e$  es la probabilidad de que un byte de un paquete no este corrupto  
 $d$  es el número de bytes del paquete

## 3. Modelos ADC

El modelo elegido especifica como se generan las lecturas cogidas del ADC. Hay dos modelos:

- Random: Cada canal devuelve un valor de 10 bits aleatorio.
- Generic: También provee valores aleatorios por defecto, pero añade funcionalidad. Hay aplicaciones externas que pueden actuar en este modelo usando el canal de control de TOSSIM, configurando el valor de cualquier puerto ADC o cualquier nodo. Actualmente, la única herramienta que lo soporta es TinyViz (herramienta java de visualización y medio de actuación para TOSSIM). Para más información consultar la referencia [12].

## 4. Modelo de concurrencia

TOSSIM modela cada interrupción de TinyOS como la simulación de un evento. Cada evento esta asociado con un nodo específico. El simulador de eventos se ejecuta automáticamente con respecto a otro evento. De esta forma, a diferencia del hardware real, las interrupciones no pueden adelantarse una a otra. Tras la ejecución de cada simulador de eventos, TOSSIM chequea la cola de tareas, y ejecuta las tareas pendientes en orden FIFO. En TOSSIM no hay tareas adelantadas. Este método de ejecución de tareas supone que las tareas recursivas pueden causar que TOSSIM ejecute estas tareas indefinidamente. Usar las tareas de esta forma es contrario al modelo de programación dirigido por eventos de TinyOS. Cuando un simulador de eventos llama a un manejador de interrupción, éste ejecuta el código a través de comandos y eventos.



## Apéndice C: Manual de NesC.

NesC 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.

### 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.

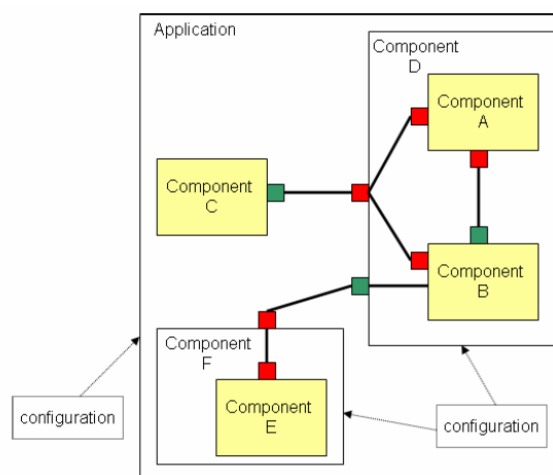


Fig. C.1: Aplicación como un conjunto de componentes agrupados y relacionados.

- 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.

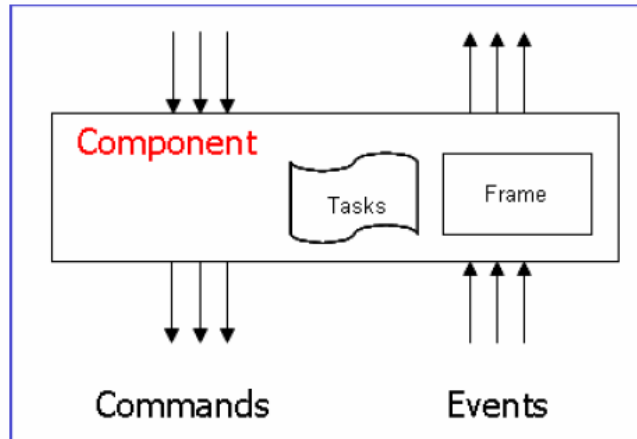


Fig. C.2: Diagrama de comandos y eventos

- 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.

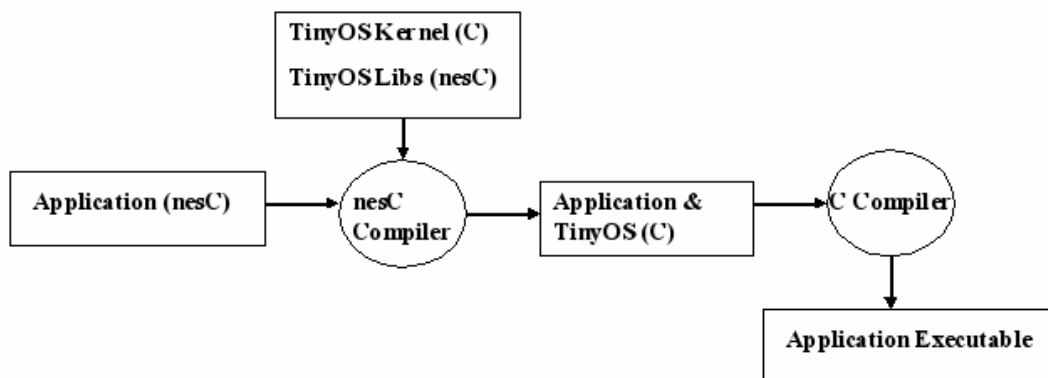


Fig. C.3: Proceso de compilación de una aplicación TinyOS.

## Tipos de Datos

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

1. **uint16\_t** es un entero sin signo de 16 BIT
2. **uint8\_t** es un entero sin signo de 8 BIT
3. **bool** es un booleano ( TRUE , FALSE )
4. **result\_t** es un booleano con los valores SUCCES y FAIL

## 1. 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.

### 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)

“*miaplicacionM.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 módulo).

### Especificación de un Componente

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

*Especificación de un componente como un módulo*

### Archivo NesC de “miaplicacionM.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;
Interface Y;}           ⇒           Uses interface X;
                                   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.

### 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.

**Resumen de la estructura**

Los ficheros de la aplicación que se ha usado de ejemplo en este apartado quedan reflejados en la *Fig. C.4*, en la que también se ha incluido un tipo enumerado “paquete”.

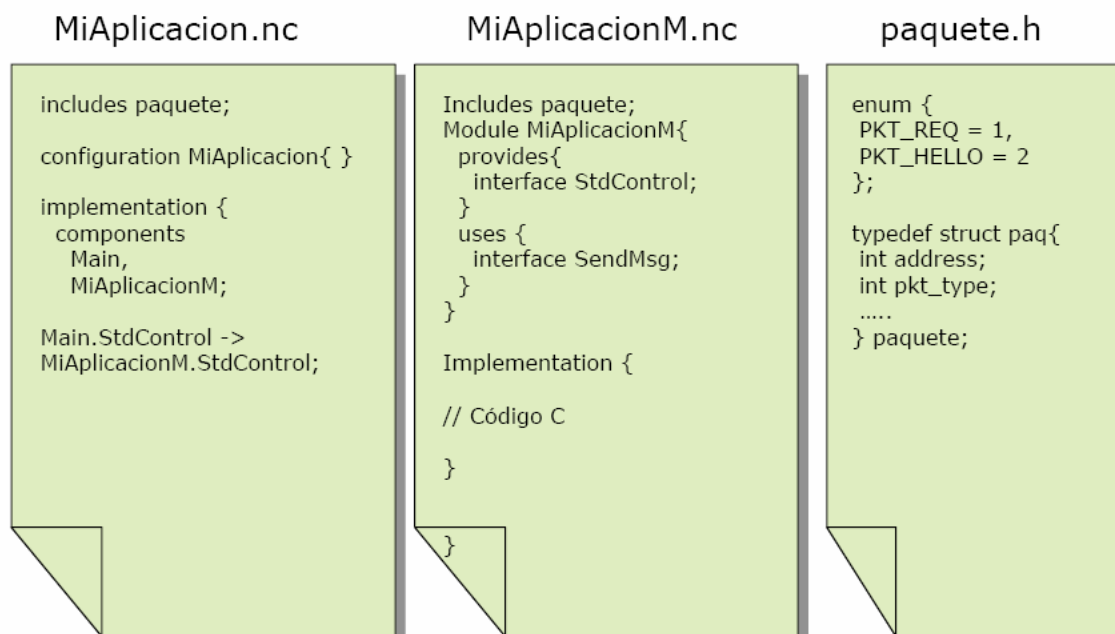


Fig. C.4: Resumen de la estructura de la aplicación de ejemplo “MiAplicacion”

**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 interconexión (*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 2* de este apéndice.

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()`.

## 2. 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.

### 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`.

## 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 sería:

```

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 varias 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);
```

## 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.

## Interfaz SendMsg

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

## 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.

### 3. 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.

#### 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 [ $e_1, e_2, \dots, e_n$ ] 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 señalado. 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 cuando 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.

## 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.

## 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`.

## 4. 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.

### 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.

### 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 señalizadores 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 embargo, 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`).

### 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 válida.

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;
  StdControl as SC;
} ...

module M2 {
  uses interface
} ...

configuration C {
  implementation {

interface X {
  command int f();
  event void g(int x);
}

module M {
  provides interface X as P;
  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.

## 5. Concurrency en NesC

NesC asume un modelo de ejecución que consiste en ejecución para completar las tareas (el cómputo en curso) y manejadores de interrupción que son señalados asincrónicamente 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 son 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 módulo (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.

## Apéndice D: Cómo portar la aplicación al nodo.

Antes de nada, para cargar una aplicación al nodo se necesita:

- Tener instalado TinyOS según se explica en el Apéndice B,
- Incluir las carpetas y ficheros con nombre `ucm_eeg_dyn` y `ucm_eeg_sta` que se indican en dicho Apéndice.
- 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, lo primero es compilar el programa que se quiere portar desde la carpeta donde están ubicados los ficheros `.nc` y `.c`. Se distinguen dos formas para compilarlo: la estática y la dinámica, cuya diferencia radica 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
```

El 1 de *install* significa que vamos a compilar el programa para ser cargado en el nodo 1. Si se quisiera cargar el programa en varios nodos, por ejemplo 3 nodos, habría que crear un binario para cada nodo, compilando tres veces cambiando ese valor a 1, 2 y 3.

Una vez compilado, el siguiente paso es abrir la carpeta `demo/test/ecg` y sustituir el archivo con llamado *main.ihex.out-1* (en el caso de haber varios nodos, este valor sería `.out-2`, y `.out-3` para los dos nodos adicionales) 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 [27]. Este programa es un Integrated Development Environment (IDE) para la compilación y la depuración de aplicaciones para el microcontrolador MSP430. 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 única forma de cargar programas en el MSP430 [26] del nodo es a través del modo debug.

Al abrir el programa para empezar a trabajar, se muestra una ventana de inicio donde puedes 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 está desarrollado exclusivamente para trabajar con el sensor de 25 canales.

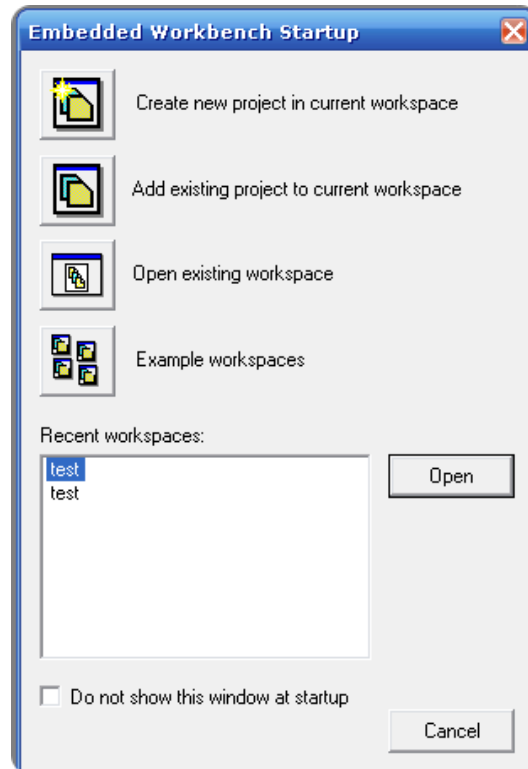


Fig. D.1: Ventana de inicio.

Dicho 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 nuestras aplicaciones haremos Project → Debug. Entonces se cargará la aplicación en el nodo, lista para ser ejecutada. En este paso puede que dé algunos errores que normalmente son causados por la conexión incorrecta del nodo, y se suelen solucionar desconectando y conectando de nuevo.

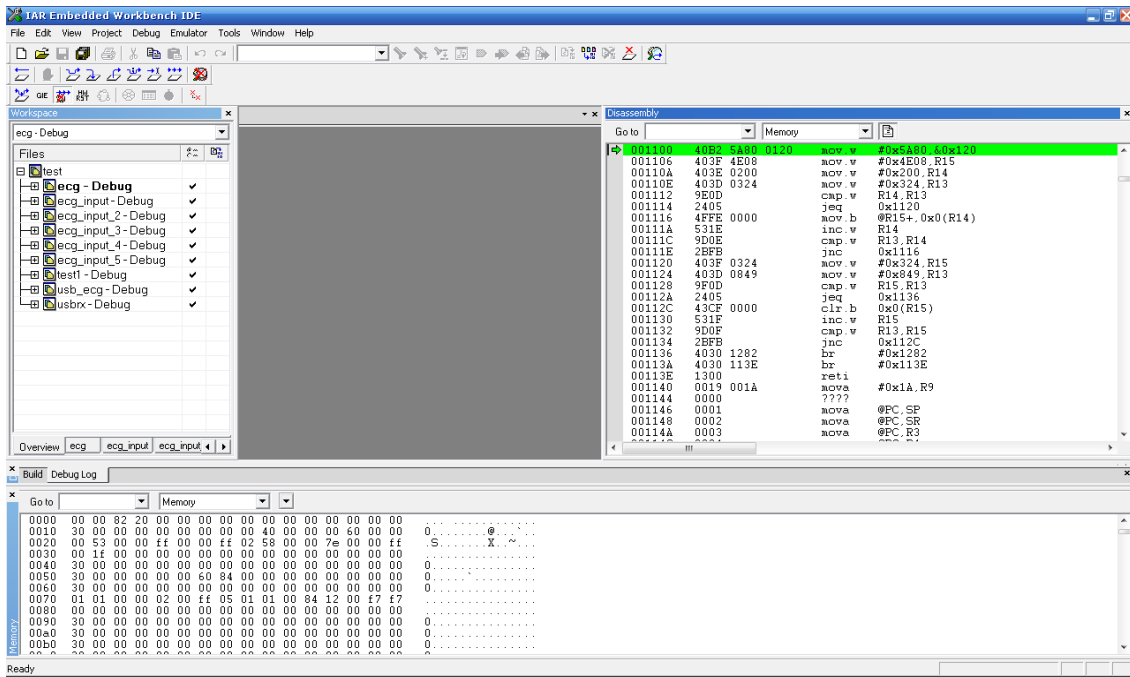





Fig. D.2: Pantalla de Debug del IAR Embedded Workbench.

En el panel inferior se muestra el mapa de memoria del nodo, en el panel derecho el código ensamblador de la aplicación. Para programas más sencillos puede ser útil tener estos paneles, pero en nuestro caso se han usado otros métodos de depuración usando los leds del nodo y la estación base.



Fig. D.3: opciones de Debug.

En la parte superior se encuentra la barra de herramientas de depuración, mostrada en la imagen D.3. Para nuestro modo de ejecución solo es necesario usar 3 botones.

-  Ejecutar. Otra posibilidad es pulsando F5.
-  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.



## Bibliografía

1. Yan Sun, Kap Luk Chan and Shankar Muthu Krishnan: *Characteristic wave detection in ECG signal using morphological transform*. BMC Cardiovascular Disorders 2005, 5:28.
2. Yan Sun, Kap Luk Chan and Shankar Muthu Krishnan: *ECG signal conditioning by morphological Filtering*. Computers 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. TinyOS: <http://www.tinyos.net/>
6. Contiki: <http://www.sics.se/contiki/>
7. MANTIS: <http://mantis.cs.colorado.edu/>
8. Nano-RK: <http://www.nanork.org/nano-RK>
9. SOS: <https://projects.nesl.ucla.edu/public/sos-2x/doc/>
10. BTnut: <http://oss.kernelconcepts.de/nutos/>
11. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler, Eric Brewer: *The nesC language: A holistic approach to networked embedded systems – PLDI: programming language design and implementation - 2003*
12. Jason Lester Hill: *System Architecture for Wireless Sensor Networks - University of California, Berkeley 2003*
13. Nordic Semiconductor (2000), *nRF2401 Transceiver Data Sheets*, <http://www.nordicsemi.com>
14. Texas Instrument, microcontrolador MSP430x149: <http://www.ti.com/>
15. Philip Levis and Nelson Lee: *TOSIM: A Simulator for TinyOS Networks - September 17, 2003*
16. Sistema Operativo para sistemas empotrados eCos: <http://ecos.sourceforge.org/>
17. Sistema Operativo para sistemas empotrados  $\mu$ C/OS: <http://www.micrium.com/>
18. Daskalov I.K. y Christov I.I. (1999): *Automatic detection of the electrocardiogram T-wave end*. Mel Biol Eng Comput, 37 (3): 348 - 353
19. Li C., Zheng C., y Thay C. F (1995): *Detection of ECG characteristic points using wavelet transforms*. IEEE Transactions on Biomedical Engineering, 42:21 - 29
20. Bert Gyselinckx, Chris Van Hoof, Julien Ryckaert, Refet Firat Yazicioglu, Paolo Fiorini, Vladimir Leonor: *Human++: Autonomous Wireless Sensors for Body Area Networks – IMEC*
21. DUBIN, D. (1976) *Electrocardiografía práctica*. , México D. F., McGraw-Hill Interamericana.
22. Culler D.: *TinyOS: Operating system design for wireless sensor networks*. Sensors, page 41 – 49 (2006)
23. Culler D., Estin D., Srivastava M.: *Overview of sensor networks*. Computer, pages 41 – 49 (2004)

24. Lo, B. and Yang, G. (2005). *Key technical challenges and current implementations of body sensor networks*. - Second International Workshop on Wearable and Implantable Body Sensor Networks.
25. Schamroth, L. (1971). *The disorders of cardiac rhythm*. Blackwell Scientific Publications.
26. MSP430 IAR Embedded Workbench™ IDE User Guide for Texas Instruments' MSP430 Microcontroller Family - Part number: U430-3
27. IAR Site: <http://www.iar.com>
28. IMEC: <http://www2.imec.be/>
29. Cygwin: <http://www.cygwin.com/>