REGULACIÓN AUTOMÁTICA MEDIANTE UNA RED DE SENSORES INALAMBRICOS DE LA INSOLACIÓN EN UN ESPACIO HABITABLE PARA CONTROL DE SU TEMPERATURA

Agustín Isasa Cuartero

Proyecto de Fin de Carrera
Área de Sistemas Empotrados
2º ciclo de Ingeniería Informática

Consultor: Sebastià Cortes Herms
Universitat Oberta de Catalunya - UOC
Junio 2012

UOC
www.uoc.edu

RESUMEN

En este Proyecto desarrollamos el prototipo de un sistema de automatización de apertura y cierre de una persiana motorizada que permita regular la insolación de un determinado espacio habitable y colaborar a la regulación de su temperatura, y que por lo tanto ayude a la consecución de la comodidad de los usuarios y a la obtención de ahorro energético. Para su implementación mediante una red de sensores inalámbricos haremos uso del siguiente conjunto de tecnologías: elementos hardware cuya tipología y funcionalidad nos permite asignarlos a la categoría de sistemas empotrados; comunicaciones entre sensores y microcontroladores; comunicaciones nodo a nodo y nodo-PC; aplicaciones software especializadas para el hardware empotrado; y aplicaciones software para ordenador de sobremesa que permitan la explotación del sistema por los usuarios. El Proyecto se constituye como el trabajo básico de la asignatura *Proyecto de Fin de Carrera*, área de *Sistemas Empotrados*, de los estudios de segundo ciclo de Ingeniería Informática en la Universitat Oberta de Catalunya - UOC.

Palabras clave: sistemas empotrados, sistemas embebidos, microcontroladores, sensores, redes inalámbricas de sensores, WSN, ZigBee, TinyOS, nesC, C, Linux, Java.

INDICE

MEMORIA PFC

	MEMORIA PFC	Página
1	**INTRODUCCIÓN**	**6**
2	**LA APLICACIÓN A DESARROLLAR**	**7**
2.1	Punto de partida en el PFC	7
2.2	Contexto del sistema a desarrollar	8
2.3	La aplicación real	9
3	**OBJETIVOS DEL PROYECTO**	**12**
4	**TECNOLOGIA**	**13**
4.1	Hardware	13
4.2	Software	14
4.3	Arquitectura de TinyOS	15
4.3.1	El modelo de ejecución	15
4.3.2	El modelo de abstracción y programación	16
4.3.3	Componentes básicos en TinyOS	17
4.3.4	Comunicaciones, arquitectura cliente-servidor e interrelación TinyOS-PC	18
4.4	El entorno de programación	22
5	**PLANIFICACIÓN DEL PROYECTO**	**24**
5.1	La planificación inicial	24
5.2	La planificación definitiva	27
6	**CASOS DE USO DE LA APLICACIÓN**	**29**
6.1	Los actores y sus funciones en la aplicación	29
6.2	Casos de uso	30
6.2.1	Actor Usuario	31
6.2.2	Actor Nodo de Control	32
6.2.3	Actor Nodo Actuador	35
7	**DISEÑO**	**36**
7.1	Punto de partida para el diseño	36
7.2	El proceso de desarrollo	37
7.3	Diseño de la aplicación	38
7.3.1	Diseño para el actor Usuario	39
7.3.2	Diseño para el actor Nodo Actuador	40
7.3.3	Diseño para el actor Nodo de Control	40
7.4	Detalle de la lógica de la aplicación	43
7.5	Adaptación del diseño a TinyOS	46
7.5.1	Nodo actuador	46
7.5.2	Nodo de control	47
7.6	La interface de usuario	49

8 IMPLEMENTACIÓN 51
8.1 Nodo de control 51
8.1.1 *GestionSensoresEntornoP* y *EnviarComandoActuadorP* 51
8.1.2 *ActualizarLogP* 53
8.1.3 *GestionSensorHallP* 54
8.1.4 *GestionParametrizacionP* 55
8.1.5 Otros elementos del código 55
8.2 Nodo actuador 56
8.2.1 *GestionSensorHallP* 57
8.2.2 *GestionMotorP* 57
8.2.3 Otros elementos del código 57
8.3 Aplicación de usuario 58
8.3.1 La interface gráfica 59

9 MANUAL DE USUARIO: DIRECTRICES 60

10 CONCLUSIONES 61
10.1 Mejoras y posible evolución futura del sistema 61
10.2 Conclusiones finales 61

11 BIBLIOGRAFIA, DOCUMENTACIÓN 63

ANEXOS

A1 UNA NOTA SOBRE EL CALCULO DE LA TEMPERATURA AMBIENTE 66

A2 MANUAL DEL USUARIO 68
INTRODUCCIÓN 68
A2-1 COMPILAR E INSTALAR LAS APLICACIONES 68
A2-1.1 Preliminares 68
A2-1.2 Compilación de la aplicación para el nodo actuador 69
A2-1.3 Compilación de la aplicación para el nodo de control y la aplicación de usuario 71
A2-1.4 Instalación del hardware 75
A2-1.5 Posibles tareas de limpieza 75
A2-2 MANUAL DE USUARIO APLICACIÓN PC *BlindControl* 77
A2-2.1 Introducción 77
A2-2.2 Arranque y primeras acciones de la aplicación de usuario 77
A2-2.3 Revisión del log de la aplicación 79
A2-2.4 Modificación de los parámetros del sistema 81
A2-2.5 Modificar el visor/editor del log 81
A2-2.6 La aplicación gráfica 82

1 - INTRODUCCIÓN

Para el área de Sistemas Empotrados del Proyecto de Fin de Carrera de los estudios de Ingeniería Informática se nos ha solicitado el desarrollo de un proyecto que debe incluir un sistema de electrónica empotrada y el correspondiente software de control y comunicación.

En particular, contamos con un hardware compuesto por dos tarjetas gemelas que incluyen sendos microcontroladores (ATMega 1281) y sistemas de comunicaciones inalámbricas *ZigBee* en la banda de los 2.4 Ghz, así como, entre otras funcionalidades, soportes auxiliares para gestión de sensores, comunicaciones con un ordenador vía USB, y entradas y salidas.

Por otra parte, como sistema operativo se nos propone *TinyOS*, un SO diseñado especialmente para sistemas empotrados y comunicaciones inalámbricas.

Con las herramientas que contamos podremos hacer frente a multitud de necesidades reales que puedan ser afrontadas con una estructura de solución que suponga:

- Diseño y compilación cruzada (normalmente en un PC) de una aplicación que gestione entradas (habitualmente desde sensores) y salidas (habitualmente hacia dispositivos analógicos o digitales).
- Descarga de la aplicación compilada en las tarjetas mediante (usualmente) una conexión serie a través del correspondiente puerto del ordenador.
- Habilitación de uno o varios sensores que registren variaciones de magnitudes físicas habituales (luz, temperatura, voltaje...), así como de una o varias salidas digitales que permitan gobernar un dispositivo físico analógico o digital, electrónico y/o mecánico. En general, este gobierno dependerá de las entradas registradas de los sensores, y corresponderá a la principal funcionalidad de la aplicación.
- Comunicación inalámbrica entre tarjetas para transferir información o comandos.

Como proyecto de aplicación genérica, vamos a considerar la gestión de un motor de corriente continua de acuerdo a la temperatura/luz captada por un sensor al efecto. Las mediciones del sensor motivarán decisiones que serán transmitidas de forma inalámbrica a otro nodo que activará, en un sentido u otro, el motor DC, que a su vez actuará para abrir o cerrar una persiana. Además, se habilitará un *log* para registro en un ordenador personal de las diferentes medidas y acciones resultantes en la actividad del sistema.

2 - LA APLICACIÓN A DESARROLLAR

2.1 Punto de partida en el PFC:

La formación de un ingeniero informático está centrada, como parece natural, más en la programación (es decir, dar estructura a la información y a las posibles modificaciones de utilidad sobre ella) que en el hardware soporte de la misma. No obstante, el actual currículum de los estudios incluye diferentes aproximaciones a la *máquina* (electrónica digital, estructura de computadores, sistemas operativos, estructura de redes...), por lo que el funcionamiento del hardware no es ajeno por completo al ingeniero en informática.

Como especialización tanto de hardware como de software, están surgiendo en los últimos años sistemas electrónicos integrados que replican en parte las funcionalidades que hasta hace algunos decenios sólo eran implementables con ordenadores completos. La generalización y abaratamiento del diseño y producción de placas base miniatura encapsulando completos mecanismos de programación, ejecución, control y comunicación, ha originado la explosión de sistemas empotrados en la que estamos inmersos, y cuya ubicuidad es evidente cuando reflexionamos sobre su uso en electrodomésticos, automóviles, máquina-herramienta, transporte, industria, arquitectura, ingeniería...

Ahora bien: un sistema empotrado no alcanza (la tentación es añadir el adverbio *todavía*) la operatividad y funcionalidad de un PC. Las limitaciones de los microcontroladores (en comparación con los microprocesadores), la gestión de la memoria, o los propios mecanismos de entrada y salida, hacen que la explotación de estos sistemas esté más cercana a la máquina que en los sistemas con microprocesador. La enorme complejidad y capacidad de computación de, por ejemplo, un equipo de sobremesa, permite unos elevados niveles de abstracción para con respecto al hardware que hacen que, una vez resueltos, la gestión y programación de estas máquinas sean relativamente sencillas.

Estos niveles de abstracción son necesariamente más limitados en los sistemas empotrados, lo que requiere una especialización en las herramientas cuyo conocimiento es, en general, muy limitado (si no nulo) en el ingeniero no especializado. Así, los lenguaje de programación (aunque fuertemente basados, generalmente, en lenguajes conocidos) y sus esquemas de programación, así como los propios sistemas operativos de uso habitual en esta electrónica embebida, son prácticamente nuevos para el autor de este PFC, aunque hay que confiar que sean de aplicación los conceptos genéricos que sobre programación, sistemas operativos, redes y hardware computacional se han llegado a trabajar durante los estudios.

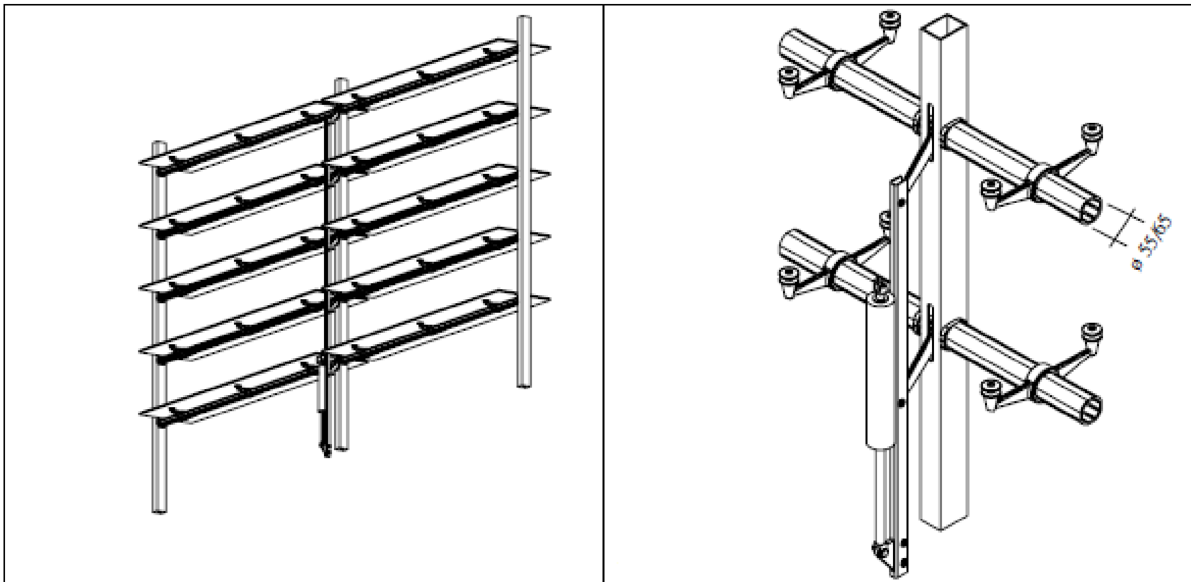
Uno de los paradigmas de implementación de estos sistemas empotrados viene teniendo particular éxito en ámbitos académicos e industriales: se trata de las redes inalámbricas de sensores, o WSN (del inglés *Wireless Sensor Network*). Consisten en redes de sensores distribuidos en un determinado espacio físico, que generalmente (y en su aproximación más básica) aportan de forma controlada información a un nodo central. Este último toma decisiones y ejecuta acciones según la información recolectada. De acuerdo a nuestros objetivos, este enfoque será el apropiada para nuestra aplicación.

2.2 Contexto del sistema a desarrollar:

La actual crisis económica, precedida por varias crisis energéticas en los últimos decenios, merece un reestudio de los métodos constructivos para, entre otros campos, intentar adecuar el consumo energético minimizándolo, pero manteniendo al mismo tiempo unas medidas adecuadas de confort para los usuario de los actuales edificios. Además, y puesto que nos encontramos en un contexto de amplísima oferta, cualquier valor añadido que el promotor de estas construcciones (edificios de oficinas, naves industriales, locales, viviendas...) pueda aportar como factor de diferenciación puede convertirse en un factor competitivo que incline definitivamente hacia sus intereses alguna parte sustancial de las decisiones de compra que puedan encontrarse en el mercado.

La gradación de la luz que se recibe por los vanos de una edificación tiene un efecto directo sobre la temperatura de un espacio. El *efecto invernadero* puede causar altas temperaturas que requieran una inversión energética elevada para ser reducidas, o un continuo tratamiento manual de persianas o cortinajes a lo largo de las diferentes horas del día para evitarlas. Por el contrario, una mínima penetración de luz natural, además de hacer incómoda la visión, puede requerir de mayor acopio de calor artificial que también aumentará el gasto energético.

En los sentidos anticipados, una instalación de persianas tipo lamas (como las que pueden observarse en http://www.colt.es/productos-y-sistemas/pdf-files/shadoglass.pdf), y cuyo movimiento pueda automatizarse de forma sencilla en función de determinados niveles de temperatura o luminosidad, parece una solución apropiada al problema que intentamos resolver.



Un ejemplo de diseño de persiana de lamas móviles

2.3 La aplicación real:

En su implementación ideal, contaríamos con un conjunto de sensores de luz/temperatura distribuidos proporcionalmente según superficie y/o volumen del espacio a monitorizar, así como el número de actuadores necesario para regular la posición de las lamas que, a su vez, regularán la cantidad de insolación del espacio.

El componente individual, o nodo, habitual de una red de sensores suele ser una placa de circuito que aloja conjuntamente un microcontrolador y un módulo de radiofrecuencia capaz de habilitar una infraestructura de red inalámbrica si se somete a una programación específica. Estas placas cuentan con conexiones de entradas y salidas que permiten su conexión con el exterior, y normalmente son capaces de manejar tanto señales digitales como analógicas. Para nuestros efectos, es posible conectar a estas placas sensores de diferentes tipos cuyas variaciones de voltaje en sus salidas (o su representación binaria tras su codificación vía conversión analógico-digital) pueden ser leídas por las entradas del microcontrolador. De manera recíproca, la placa puede ofrecer salidas (habitualmente digitales) utilizables para controlar periféricos a través de relés o circuitos transistorizados de amplificación de potencia.

Para implementar nuestra aplicación real vamos a contar con dos nodos gemelos (o *motes*) que ya cuentan, de forma integrada, con tres sensores diferentes (luz, temperatura y efecto Hall) cada uno, así como tres salidas explícitas (de entre otras muchas genéricas) cuyos estados son fácilmente apreciables al alimentar sendos *leds*, también integrados en cada placa.

A pesar de la absoluta identidad física entre las dos placas, su utilización resulta completamente diferente puesto que distribuimos de forma asimétrica entre los dos nodos las funcionalidades de medición, control, actuación y comunicación. Como vemos en el diagrama básico de la instalación, acomodaremos un nodo en la persiana de lamas (en lo sucesivo lo denominaremos como *nodo actuador*) con el objetivo de actuar tanto en un sentido como en otro sobre el motor de oscilación de las mismas, incluyendo, además, una función de detección en cada mecanismo de lamas (aprovechándonos del sensor de efecto Hall) que determina si está completamente cerrada o no. Por otra parte, confiamos al otro nodo las labores de comprobación de temperatura y luminosidad, así como la implementación de la lógica de apertura o cierre de las persianas en función de las medidas de los sensores (en adelante lo denominaremos *nodo central* o *nodo de control*). Los dos nodos se comunican entre sí (intercambiando medidas de los sensores o comandos) mediante sus módulos de radio, y el nodo central también establece comunicación con el PC para labores de parametrización y de *log*. De esta forma, cada pareja de nodos será una unidad mínima de control, actuación, programación y gestión compuesta de las dos placas, y su extensión a un mayor número de actuadores no requeriría una excesiva complejidad.

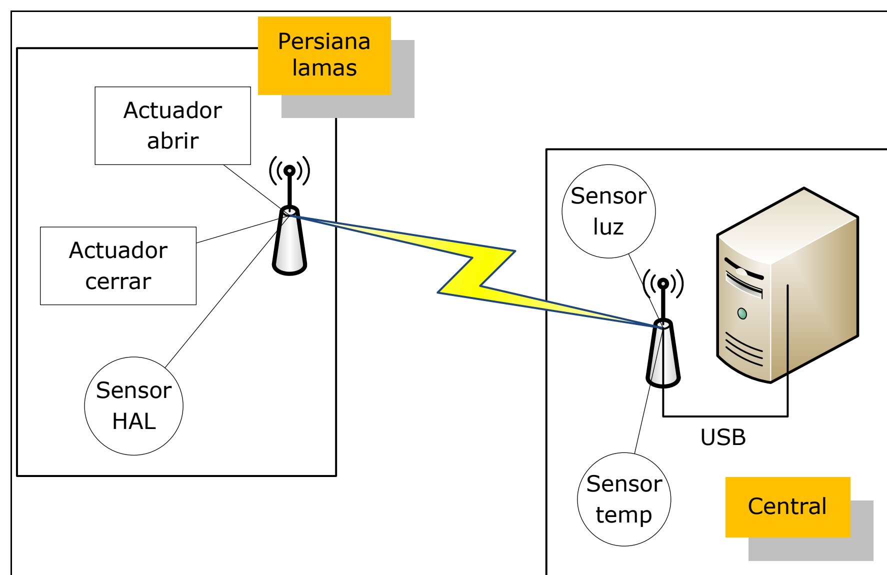


Diagrama hardware de la instalación

Mientras que el nodo actuador debe, fundamentalmente, ejercer acciones sobre las lamas de acuerdo a la lógica de la aplicación (que actuará según las lecturas recibidas de los sensores), el nodo central o de control se dedicará a las actividades de medida de los sensores, a procesarlas, y a comunicar los comandos al nodo actuador de forma efectiva. Este nodo central, por tanto, será el que implemente la lógica de la aplicación. En cualquier caso, y puesto que esta lógica deberá ser modulada según voluntad del usuario, será necesaria una aplicación para ordenador de

sobremesa, o PC, que permita modificar los parámetros básicos de las acciones programadas, por lo que deberá contarse con la oportuna comunicación entre ordenador y *mote*, en este caso mediante cable y a través del correspondiente puerto serie (aunque implementado mediante una conexión USB estándar).

De esta forma, y de acuerdo a los requisitos planteados, vamos vislumbrando un diseño preliminar y básico de nuestra aplicación, que podemos describir mediante la enumeración, con una descripción informal, de sus requisitos iniciales:

- Un sensor de temperatura y otro de luz (conectados al nodo central) recogen periódicamente datos de su entorno, que son procesados en este mismo nodo central según una lógica prefijada.
- En su caso, este nodo central transmite, también periódicamente y mediante un canal inalámbrico, instrucciones a la placa actuadora para que gobierne el motor asociado a la persiana.
- Este actuador controla mediante el motor un conjunto de lamas, que se abrirán para permitir pasar la luz exterior cuando la temperatura baje de un determinado umbral, así como impedirá el paso de la luz directa cuando la temperatura supere otro umbral preestablecido. Los resultados serán modulados, también, por la luminosidad medida en el entorno. El objetivo es abrir o cerrar las lamas para permitir una mayor o menor insolación y facilitar así la regulación de la temperatura.
- La fijación de estos umbrales, así como otros posibles parámetros necesarios, serán transferidos a la aplicación del nodo de control desde un PC, y serán introducidos en la correspondiente aplicación por el usuario.
- Cada mecanismo de lamas mantiene un añadido magnético que permitirá, a través del sensor de efecto Hall, conocer si el conjunto de lamas está cerrado completamente o no.
- El nodo de control transferirá al PC tanto las lecturas efectuadas por los sensores como las acciones llevadas a cabo por el sistema (apertura y cerrado de las lamas), información que será registrada en un log manejado al efecto por la aplicación en el PC.
- Es necesaria, por tanto, una aplicación en el PC, que ha resultado adecuado programarla en Java, que habilite, en resumen, dos funcionalidades básicas:
    - La parametrización de los efectos del nodo actuador de acuerdo a las medidas de los sensores y
    - la habilitación de un mecanismo de registro de medidas y actuaciones.

3 - OBJETIVOS DEL PROYECTO

No puede perderse de vista el carácter académico del proyecto que, junto con lo limitado de la infraestructura hardware (no existe un mecanismo real sobre el que actuar), prevalece necesariamente sobre posibles objetivos *reales* (modificación de una determinada realidad física mediante una infraestructura basada en un sistema empotrado).

Como objetivo académico principal, y a caballo entre el ámbito teórico y los posibles proyectos a desarrollar bajo una dimensión física, este trabajo de aplicación de las asignaturas de ingeniería del software, por un lado, y de proyectos, por otro, debe permitir:

- Desarrollar y comprobar las capacidades de abstracción para el análisis, diseño e implementación de una aplicación real.
- Desarrollar y comprobar las capacidades de análisis, ejecución y control de las dimensiones temporales, de calidad y de resultados de un proyecto (obviamos en esta ocasión, a pesar de su importancia, la dimensión económica).
- Desarrollar y comprobar las capacidades de síntesis y comunicación para documentar y trasladar a terceros los puntos anteriores.

Además, y originados en el área PFC elegida, encontramos otros objetivos formativos que son de aplicación práctica inmediata:

- Comprensión de los fundamentos de los sistemas empotrados y de las WSN, o redes inalámbricas de sensores, y ello en las diferentes dimensiones y áreas que pueden identificarse: programación, comunicación, computación, actuación.
- Comprensión y explotación de los componentes hardware de los nodos: microcontrolador, memoria, radio, sensores, UART, IO...
- Comprensión y explotación de la infraestructura software de los nodos y de un ordenador PC común, y en particular del sistema operativo de elección (TinyOS).
- Reflexión sobre las amplias posibilidades de uso de este tipo de sistemas empotrados y de las WSN.

Por último, y a pesar de lo apuntado en el inicio de este apartado, no debemos olvidar que subyacen en este proyecto unas posibilidades claras de aplicación práctica:

- Desarrollo y simulación de un sistema automatizado para la regulación de la insolación a través de un vano en un espacio construido, mediante la motorización de un mecanismo de lamas movibles controlado por una aplicación que toma decisiones de acuerdo a mediciones de temperatura y luz ambiente.

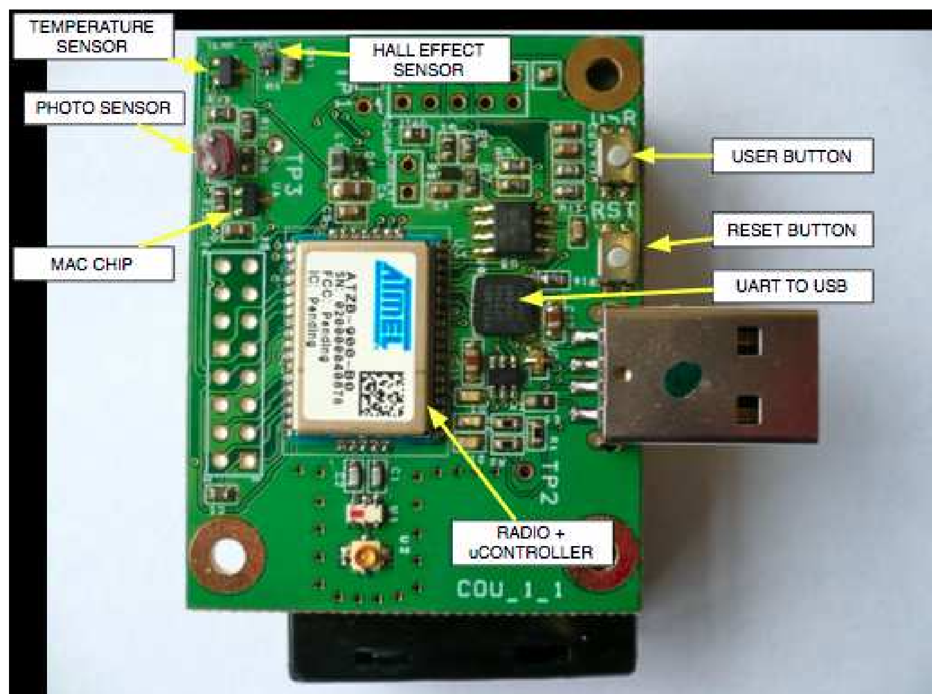
4 - TECNOLOGÍA

La tecnología de aplicación para nuestro proyecto nos viene dada, en buena medida, por las placas hardware, o *motes*, de las que vamos a disponer. En consecuencia, el hardware queda absolutamente definido, al igual que el sistema operativo de referencia para la gestión de las placas.

Será en la aplicación que corra sobre el PC donde ha existido cierto grado de libertad y ha habido que efectuar elección de tecnología software.

4.1 Hardware:

Para el desarrollo del proyecto contamos con dos placas gemelas cuya dotación básica está caracterizada fundamentalmente por la integración de un microcontrolador ATmega1281 y un módulo de radiofrecuencia AT86RF212, ambos del fabricante Atmel Corporation. La propia Atmel proporciona un módulo integrado con ambos componentes denominado ATZB-900-B0.



El hardware COU_1_1, que incluye el módulo ATZB-900-B0

El ATmega1281 es un microcontrolador de propósito general con arquitectura RISC de 8 bits, que combina memorias flash, SRAM y EEPROM, y que cuenta con un importante número de entradas y salidas que le dotan de una gran flexibilidad. Alcanza los 8 MIPS a 16MHz (su

frecuencia máxima de operación), y su voltaje mínimo de 2,7 V le proporciona un consumo moderado.

En cuanto al módulo de radiofrecuencia AT86RF212, trabaja en la banda de los 2,4 GHz, y se adhiere, entre otros, al estándar *ZigBee* (basado en el IEEE 802.15.4). Su función principal es establecer una interface radio completa entre la antena y el microcontrolador, y se fabrica como un módulo completamente integrado con la excepción de la propia antena, el oscilador, y algunos componentes de filtrado.

Las placas que nos ocupan vienen también con tres sensores integrados: uno de luminosidad, otro de temperatura, y un último sensor de efecto Hall para detectar variaciones en campos magnéticos.

Por último, estas *motes* también proporcionan interruptores (de reset y programable), varios *leds* conectados a algunas de las salidas, así como un completo circuito de comunicaciones serie con componentes UART, si bien encapsulado bajo una conexión USB para facilitar la conexión del módulo con los ordenadores modernos.

Vemos, pues, que se trata de un hardware adecuado para la implementación de nuestro proyecto: por un lado, contamos con los sensores necesarios para nuestros propósitos; por otro, las necesidades tanto de comunicación inalámbrica como con el PC se resuelven por las facilidades del módulo RF y del subsistema UART/USB, respectivamente; finalmente, el microcontrolador ATmega1281 constituye una plataforma idónea para implementar la lógica requerida. Es de reseñar que aprovecharemos los *leds* como simuladores de las salidas (al motor que mueve las lamas, por ejemplo).

4.2 Software:

Uno de los principales recursos que caracteriza a cualquier sistema empotrado es el sistema operativo que subyace en él. Entre los diferentes sistemas operativos para este tipo de electrónica, destaca por sus funcionalidades TinyOS, desarrollado en los últimos años en la Universidad de California en Berkeley y especialmente diseñado para dar soporte a redes inalámbricas de sensores, o WSN.

Aunque en el siguiente apartado, y dada su trascendencia, ampliamos los detalles de este sistema operativo, avanzamos que los modelos de ejecución y abstracción de TinyOS se caracterizan entre otras cualidades, por:

- Orientación a eventos, que son disparados mediante la activación de interrupciones originadas por temporizadores, pulsadores u otro hardware. Un evento pospone a una tarea (no así lo contrario),

por lo que el tratamiento de los eventos debe ser lo más rápido posible.
- Optimización del uso de las limitadas memorias de los sistemas empotrados.
- Exige a las aplicaciones (escritas en nesC) su construcción mediante componentes que son enlazados mediante interfaces, permitiendo unos importantes niveles de abstracción para con respecto al hardware (incluyendo, por supuesto, el de comunicaciones).
- Habilita I/O asíncrona (o no bloqueante), y permite una alta concurrencia con una única pila, aunque a costa de una mayor complejidad de programación al tener que contemplar la integración de múltiples eventos de reducido tamaño.
- Elevado nivel de abstracción que pretende la mayor independencia posible de la plataforma micro/radio, aunque a costa de cierta pérdida de rendimiento.

El módulo integrado ATZB-900-B0 que hemos visto en el apartado anterior disfruta de un puerto para TinyOS diseñado por el ETH de Zurich. Es una implementación bien testada, aunque aparentemente su desarrollo no tiene continuidad hoy por hoy en el ETH.

Fijado el sistema operativo de referencia para nuestro sistema, quedan también fijados automáticamente otros componentes software:

- El compilador nesC, basado en componentes y orientado a eventos, es el necesario para desarrollar las aplicaciones a correr en TinyOS.
- Por su parte, la descarga a las *motes* con ATZB-900-B0 de las aplicaciones una vez compiladas se encomienda a meshprog, un software de la austriaca RSA que sustituye a los tradicionales `'make` *`plataforma`* `install'` de otras plataformas.

En cuanto al software a utilizar en el PC, se ha elegido desarrollar la aplicación mediante Java por la cantidad de información y ejemplos que para este lenguaje ofrece la documentación de TinyOS. Este entorno está completamente soportado como infraestructura de las comunicaciones entre un PC y una *mote* vía UART/USB.

Finalmente, el desarrollo del software se ha realizado bajo una distribución Linux/Ubuntu corriendo bajo una máquina virtual de VMware en Windows Vista, y dada su relativa sencillez no se ha utilizado ningún entorno de programación en particular.

4.3 Arquitectura de TinyOS:

4.3.1 El modelo de ejecución

Dadas las estrategias elegidas para acomodar el sistema operativo al hardware objetivo, TinyOS es un sistema orientado a eventos, que pueden ser disparados por interrupciones. De esta manera se puede dar servicio a los posibles y variados estímulos que pueden recibirse por los diferentes puertos de entrada que caracterizan a la generalidad de los nodos utilizados en las WSN.

Más importante todavía en términos del modelo de ejecución, el mecanismo de eventos permite que puedan eliminarse las retenciones de recursos durante amplios periodos de tiempo que son características de la ejecución síncrona de software.

Para lograr esta última capacidad e implementar la concurrencia, la ejecución de las instrucciones, en general, se deja en manos de tareas (o *tasks*; hay que precisar que en su versión nativa TinyOS no maneja hilos de ejecución), a las que se encomiendan las tareas más largas de las aplicaciones. Los eventos posponen la ejecución de las tareas para que puedan atenderse las interrupciones y, puesto que las operaciones quedan interrumpidas, estos eventos deberían ser tan rápidos como fuera posible. Las tareas pendientes se organizan en el sistema bajo una cola FIFO, y cada una de ellas se ejecuta de principio a fin antes de planificar la ejecución de la siguiente tarea.

De forma ideal, la programación debería dividir un cómputo largo en pequeños trozos de código para ser ejecutados uno tras otro. Así mismo, la capacidad de posponer tareas para más adelante permite manejar listas de ejecución priorizadas según su ordenamiento temporal.

Esta posposición de una tarea puede llevarse a cabo desde un comando (sinónimo de función; ver a continuación el modelo de programación), un evento, o desde otra tarea. De forma simétrica, una tarea puede invocar un comando o activar (*signal*) un evento.

Podríamos decir que, en general, a las tareas se les encomienda la computación, mientras que los eventos y los comandos realizan transiciones de estados.

4.3.2 El modelo de abstracción y programación

Los diseñadores de TinyOS escogieron un alto nivel de abstracción buscando la mayor independencia posible de las plataformas hardware a las que va destinado. Estas plataformas, normalmente, están compuestas de una pareja microcontrolador-radio, elementos usuales y básicos de los nodos de cualquier red WSN.

El elemento básico de abstracción y programación es el componente, que se comunica mediante interfaces con otro u otros componentes. Esta estructura de programación permite minimizar la cohesión del código gracias a su distribución, mientras que facilita la modularidad y reusabilidad de los propios componentes en la programación.

La implementación de este modelo queda a cargo del lenguaje nesC, íntimamente ligado a TinyOS, y que extiende el lenguaje C con las características de modularidad e intercomunicación entre componentes necesarias para conformar la abstracción. También incluye soporte para concurrencia vía las tareas que hemos visto en el apartado anterior, aunque recordemos que estamos ante un sistema que carece de hilos de ejecución en su concepción nativa. Estos hilos pueden añadirse, no obstante, mediante librerías.

Cada componente puede proporcionar interfaces a sus funcionalidades, que podrán ser usadas por otros componentes que las necesiten para ir así construyendo un programa de forma modular. Estas interfaces pueden diseñarse como bidireccionales, especificando un conjunto de comandos (funciones que implementa el componente que ofrece la interface) y un conjunto de eventos (funciones a implementar por el usuario de la interface). De esta manera, si un componente invoca los comandos de una interface (lo que se efectúa mediante la palabra reservada `call`), debe implementar por su parte los eventos de esa interface. El conjunto de interfaces que proporciona y usa un componente se denomina su *firma* (*signature*).

Existen dos tipos básicos de componentes en nesC: módulos y configuraciones. Los módulos implementan las funciones de las interfaces, mientras que las configuraciones indican al compilador qué componentes comunican con otros determinados componentes conectando sus interfaces (lo que en este contexto se denomina *cableado* -*wiring*-).

4.3.3 Componentes básicos en TinyOS

El acceso a los recursos hardware de cada *mote* viene explicitado en TinyOS abstrayendo, en general, cada recurso mediante un componente-módulo (o distribuyéndolo en varios si su complejidad lo requiere). Los más significativos y comunes, y que empleamos abundantemente en nuestro proyecto, son:

- **Control de leds**: el componente `LedsC` permite gestionar el encendido y apagado de los *leds* que puede incorporar el nodo hardware.

- **Temporizadores**: `TimerMillic` es un componente genérico, del que pueden instanciarse varios objetos simultáneamente, y que soporta la habilitación de temporizadores en el sistema. Como unidad de temporización emplea el milisegundo.

- **Comunicación radio entre motes**: el subsistema de radio puede gestionarse, en general, con tres componentes: `ActiveMessageC`, `AMSenderC`, y `AMReceiverC`.

- **Comunicación serie PC-*mote***: la comunicación a través del puerto serie entre una *mote* y un ordenador base se realiza mediante una estructura semejante, en cuanto a abstracción, a la comunicación por radio e involucra, por lo tanto, a los siguientes componentes: `SerialActiveMessageC`, `SerialAMSenderC`, y `SerialAMReceiverC`.

- **Servicio de interrupciones, lecturas ADC e IO genéricas**: las diferentes implementaciones de TinyOS sobre los diversos hardwares existentes intentan abstraer en lo posible la gestión de bajo nivel de la máquina. No obstante, algunas de estas implementaciones todavía no han cubierto la totalidad de los recursos disponibles en una *mote*, por lo que algunos deben tratarse a bajo nivel mediante las interrupciones y/o mediante lecturas y conversiones analógico-digitales. Puesto que la implementación de la plataforma cou24 no cubre la abstracción para los sensores, en este trabajo debemos utilizar los siguientes componentes: `HplAtm128InterruptC` (que proporciona acceso a los pins de interrupción del microcontrolador); `HplAtm128GeneralIOC` (que proporciona acceso a los pins de entrada y salida -IO- genéricas del microcontrolador); y `AdcReadClientC` (que virtualiza la capa independiente de hardware -HIL- del sistema y permite las lecturas de los sensores).

- **Secuencia de arranque**: los componentes del sistema TinyOS no sólo se vinculan a recursos hardware particulares, sino que existen otros que soportan el marco de ejecución del sistema o desempeñan funciones auxiliares. En particular, `MainC` es la interface del sistema a la secuencia de arranque, al planificador, y a los recursos hardware, por lo que este componente será necesario en nuestros programas.

4.3.4 Comunicaciones, arquitectura cliente-servidor e interrelación TinyOS-PC

TinyOS mantiene una estructura de componentes e interfaces equivalentes para sus dos mecanismos de comunicación principales:

radio y puerto serie. Para ambos se proporciona una abstracción relevante para los mecanismos de comunicación, el buffer de mensajes `message_t`, que se implementa como una estructura de nesC (del tipo `nx_struct`, semejante a las estructuras de C). `message_t` es un tipo abstracto de datos que el sistema se ocupa de instanciar en los procesos de comunicación.

`message_t` dispone de un campo `data` para indizar la carga del mensaje, que también es conveniente representar mediante otra estructura para facilitar la gestión de su contenido, evitando así tener que trabajarlo a nivel de bytes.

Una última abstracción permite el acceso multiplexado a las comunicaciones, y TinyOS la proporciona mediante la capa *Active Message* (AM). De esta manera, se proporciona a los paquetes (*packets*) que contienen los mensajes campos AM que permiten especificar las direcciones AM tanto del nodo de origen como de destino. En general, y aunque su uso es opcional, utilizaremos el más completo mecanismo AM para poder virtualizar los sistemas de comunicación del proyecto.

De esta manera, una comunicación (por radio o por puerto serie: la diferencia de nomenclatura está en el prefijo '`Serial`' para los componentes encargados de la comunicación serie) que permita el envío y recepción de paquetes puede implementarse con los siguientes componentes e interfaces:

- **`SerialAMSenderC`** o **`AMSenderC`**: componente que recoge la abstracción para el envío de mensajes AM, y que implementa una cola de paquetes que no colisiona con las colas de otros posibles *enviadores*. Proporciona, entre otras, las interfaces `AMSend` (para enviar paquetes), `AMPacket` y `Packet` (*accesores* para el contenido de los mensajes, diferenciados por su adhesión a AM o no, respectivamente).

- **`SerialAMReceiverC`** o **`AMReceiverC`**: componente que recoge la abstracción para la recepción de mensajes AM. Proporciona, entre otras, la interfaz `Receive`, que gestiona la recepción de paquetes.

- **`SerialActiveMessageC`** o **`ActiveMessageC`**: componente que implementa la capa AM para la correspondiente plataforma. Proporciona, entre otras, la interface `SplitControl`, que gestiona el encendido o apagado de los dispositivos de comunicación (radio o puerto serie).

Por razones no del todo claras, la documentación y códigos de ejemplo de TinyOS utilizan los tres componentes para proporcionar las cinco interfaces cuando el último, `SerialActiveMessageC`, las implementa en

su totalidad. Seguiremos la implementación mediante los tres componentes por su mayor claridad en el espacio de nombres y por su probada eficacia en las comunicaciones.

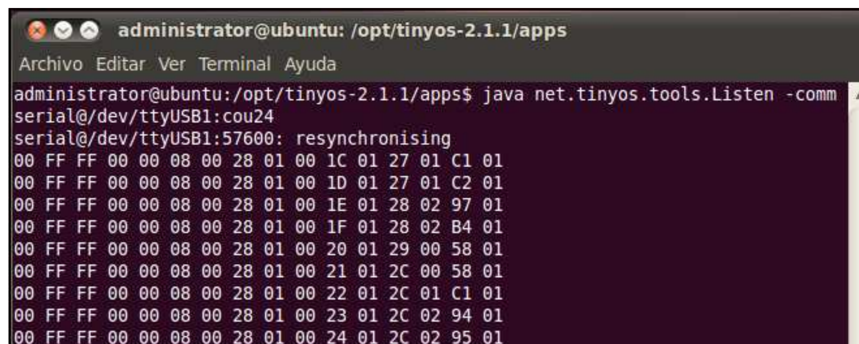
Una vez vistos los fundamentos de los subsistemas de comunicación en TinyOS, pasemos a precisar la interrelación entre una mote programada bajo este sistema operativo y un PC.

Tendremos que matizar previamente que el término *cliente-servidor*, que apunta a la arquitectura clásica en computación distribuida según la cual se distribuyen las tareas entre los nodos que proporcionan recursos (servidores) y los que los necesitan para implementar su parte de la aplicación (clientes), se relativiza cuando nos encontramos en el contexto de una WSN o, en particular, ante una aplicación distribuida en unas cuantas *motes* con una posible conexión a un ordenador. No debemos caer en la tentación de, automáticamente, considerar al PC como servidor dada su enorme capacidad de computación en comparación con un nodo. Más bien al contrario, los recursos (distribuidos) de los nodos actuarían en cierta manera como servidores de información que puede terminar por ser recolectada y gestionada por el PC cliente. Pero también cabe apreciar interacciones en sentido contrario cuando el PC programa un nodo, parametriza la aplicación distribuida, o envía comandos o instrucciones a la red. La relación cliente-servidor ahora ya no se identifica y se vincula claramente entre los distintos elementos de la red, sino que habría que buscarla en la lógica de la aplicación distribuida, identificándola más en las vinculaciones originadas por las características de los casos de uso y su implementación, que en un nodo u otro o en el PC.

En cuanto a la interrelación entre las *motes* y un ordenador, hemos visto la capacidad de habilitar con TinyOS una comunicación a través de un puerto serie con unas pocas interfaces. Tan sólo deberemos especificar en el PC el puerto serie (o USB) al que está conectado el nodo y su velocidad en baudios para tener acceso a los mensajes enviados por una *mote* conectada. La especificación de estos valores para las aplicaciones del entorno TinyOS (*tool chain*) puede hacerse bien pasándolos explícitamente como parámetros a la aplicación lectora, bien registrándolos en el sistema mediante la variable de entorno `MOTECOM`.

Pero debemos reflexionar sobre la dificultad del tratamiento de un flujo de bytes que puede obtenerse por el puerto serie del PC (o a través del puerto USB que envuelve una comunicación serie) cuando se le conecta la salida de un nodo. Resultados como los obtenidos por programas de escucha como *Listen* en el puerto serie cuando se le conecta una aplicación TinyOS como *BaseStation* o cualquiera que vuelque mensajes hacia el puerto serie del PC, nos demuestran que el formateo de tales

flujos de bytes es obligatorio para cualquier presentación que pretenda ser entendible por un usuario final:



En esta pantalla de ejemplo que utilizamos, el último byte de cada línea es un representativo de un comando, los dos bytes previos son una medida de luminosidad, los dos bytes previos son una medida de temperatura...

Para evitar una trabajosa programación de este flujo de bytes, siempre sujeta a una alta probabilidad de errores en el *parseo* de la estructura de la información, el contexto de TinyOS proporciona la herramienta MIG (*Message Interface Generator*). MIG permite la creación automática de código que será capaz de leer el flujo de bytes y extraer de él, siempre de acuerdo a un formato prefijado, la información y contenidos estructurados para su presentación clara y prácticamente sin necesidad de tratamientos adicionales para una directa comprensión por el usuario. Para su utilización se le pasan cuatro argumentos principales: el lenguaje en el que será generado el *parser*, el fichero en el que se encuentra la estructura representativa del formato de la información (ya hemos anticipado que es recomendable representar el contenido del mensaje como una `struct`, de nesC en este caso), el nombre de esta estructura, y el nombre del programa que constituirá el *parser*. Es usual, así mismo, pasarle como plataforma de destino la instrumental `null`.

Una vez compilado bajo el lenguaje de elección (permite Java, C y Python), y asumiendo un flujo de bytes de información estructurada de acuerdo al formato pasado a MIG, la ejecución de una herramienta adecuada de la *toolchain* (por ejemplo, el programa java de TinyOS `net.tinyos.tools.Listen`) a la que se le pase como parámetro el programa obtenido, volcará a pantalla el nombre de los campos de la estructura de cada mensaje junto con sus respectivos valores, obteniendo por tanto una salida legible y bien formateada.

Además del flujo de bytes indiferenciado y el estructurado gracias a MIG, tenemos una tercera posibilidad de interacción *mote*-PC: el programa bajo interfaz gráfica *SerialForwarder* que, a diferencia de los anteriores casos, permite la conexión de multitud de clientes vía TCP/IP al flujo

recibido por el puerto serie. Dado el protocolo de conexión utilizado, las aplicaciones pueden conectarse a través de Internet. En cualquier caso, las aplicaciones que se conecten a *SerialForwarder* deberán contar con sus propios medios de interpretación del flujo obtenido finalmente (puede conectarse, por ejemplo, la herramienta java `net.tinyos.tools.Listen` vista anteriormente con su *parser* complementario).

Para nuestros efectos, haremos uso de MIG (bajo java) para facilitar la interpretación y tratamiento de la información a recibir en el PC del usuario desde el nodo de control.

4.4 El entorno de programación:

Como ya ha quedado claro, el entorno de programación lo constituye la cadena de herramientas, o *toolchain*, asociada al sistema operativo TinyOS. Dados sus fundamentos *POSIX* existen dos formas básicas de ejecución de este entorno: bajo Windows pero con el soporte de Cygwin, o directamente bajo un entorno *Unix-like* como Linux. Con Windows Vista he tenido problemas en la habilitación del puerto serie, que se reconocía pero que no era capaz de establecer la comunicación entre PC y *mote*. La instalación de la imagen de una distribución Ubuntu con TinyOS instalado para una máquina virtual VMware, descargable desde la propia UOC, fue la solución definitiva para poder trabajar con cierta comodidad con el entorno.

No obstante, aún hubo que afinar el sistema para una ejecución fluida:

1. En el fichero de configuración `tinyos.sh` (que se encuentra en el directorio raíz de la instalación TinyOS) es necesario modificar el contenido de la variable de entorno `CLASSPATH` para que el sistema pueda referenciar el propio directorio de trabajo (añadiendo '.') y el contexto Java que se encuentra en el fichero `tinyos.jar`:
   ```
   ...
   CLASSPATH=$CLASSPATH:$TOSROOT/support/sdk/java:$TOSROOT
   /support/sdk/java/tinyos.jar:.
   ...
   ```
2. En el fichero `BaudRate.java` (en `support/sdk/java/net/tinyos/packet` de la instalación TinyOS, incluir una entrada para la plataforma cou24 con su velocidad de transmisión por defecto (57600 bps):
   ```
   ...
   Platform.add(Platform.x, "cou24",57600);
   ...
   ```
3. Recompilar `tinyos.jar` para que incluya el cambio anterior.

4. Modificar, de nuevo, `tinyos.sh` para incluir y exportar la variable de entorno `MOTECOM`, que ahora ya podrá hacer referencia a la plataforma cou24 como velocidad correcta:

```
...
MOTECOM = "serial@/dev/ttyUSB0:cou24"
...
```

A partir de ese momento, el entorno resulta cómodo y productivo en su manejo, aunque es frecuente durante las pruebas el cambio de dirección del puerto serie, que debe ser entonces especificado manualmente.

Las herramientas principales son:

- **Editor de código**: cualquiera. Eclipse proporciona soporte prácticamente completo para el lenguaje nesC, aunque el tamaño de nuestra aplicación hace que cualquier editor de código resulte apropiado.

- **Compilador cruzado**: `nescc/ncc/gcc/avr-gcc`, invocado para la correspondiente plataforma (cou24 en nuestro caso) a través del mecanismo `make`.

- **Programador**: `meshprog`, software de descarga de los ejecutables a las *motes*, precargadas de fábrica con un *bootloader* para evitar la necesidad del programador hardware.

El flujo de trabajo con estas herramientas es el siguiente:

- Una vez editado el código y habilitado el correspondiente archivo `Makefile`, se compila desde el directorio de las fuentes mediante `make cou24`.
- Encontraremos la imagen ejecutable en el fichero `main.srec`, en el directorio `build/cou24` creado en el directorio de las fuentes.
- `main.srec` se descarga a la *mote* mediante `meshprog`. El comando incluye el puerto de comunicación y la localización del fichero con la imagen a descargar. Por ejemplo, y en nuestro sistema: `meshprog –t /dev/ttyUSB0 –f ./build/cou24/main.srec`
- Una vez descargada, el nodo hardware comenzará automáticamente la ejecución de la aplicación.

5 - PLANIFICACIÓN DEL PROYECTO

Nos hemos encontrado, en realidad, con un proyecto dentro de otro proyecto. En un nivel global tenemos un proyecto académico que supone un conjunto de actividades tendentes al objetivo principal de culminar con éxito el hito que supone la obtención del título de Ingeniero. Para ello, debe realizarse con éxito un proyecto de aplicación real subsumido en el anterior. La planificación de uno está íntimamente ligada a la del otro, y puesto que contamos con la referencia temporal que supone el calendario de la planificación académica de la asignatura PFC, usamos ésta como marco referencial del proyecto práctico.

Así, identificamos fácilmente cinco hitos prácticamente inamovibles:

1. 23.03.2012: PEC 1, que incluye los requisitos y análisis de la aplicación, así como su planificación.
2. 24.04.2012: PEC 2, que incluye el diseño y parte de la implementación.
3. 22.05.2012: PEC 3, que incluye el resto de la implementación.
4. 07.06.2012: Entrega de la memoria final.
5. 14.06.2012: Entrega de la presentación del proyecto.

Para la planificación temporal, por tanto, hemos tenido que encajar las diferentes tareas del proyecto práctico en la guía que nos supone el calendario exigido.

5.1 La planificación inicial:

Para la PEC 1 la distribución de tareas consistió en:

	Nombre de tarea	Durac	Comienzo	Fin	Predec
1	**PEC 1 - Planificación, requisitos, análisis, preliminares**	**6 días**	**mar 20/03/12**	**dom 25/03/12**	
2	Requisitos y análisis preliminares - PEC1	6 días	mar 20/03/12	dom 25/03/12	
3	Entorno programación/compilación	6 días	mar 20/03/12	dom 25/03/12	
4	Documentación y tests TinyOS/nesC	6 días	mar 20/03/12	dom 25/03/12	
5	Redacción informe (PEC 1)	4 días	jue 22/03/12	dom 25/03/12	

Para la PEC 2, que ya incluía tareas de programación, la previsión temporal de las actividades fue la siguiente:

	Nombre de tarea	Durac	Comienzo	Fin	Predece
6	**PEC 2 - Diseño y primera entrega código**	**30 días?**	**lun 26/03/12**	**mar 24/04/12**	**2**
7	Diseño y documentación del proyecto	30 días	lun 26/03/12	mar 24/04/12	2
8	Descargas código ejemplo a motes	4 días	lun 02/04/12	jue 05/04/12	
9	Revisión documentación y pruebas código lectura sensores + encendido leds	3 días	vie 06/04/12	dom 08/04/12	8
10	Revisión documentación y pruebas código envío mensajes entre motes	3 días	lun 09/04/12	mié 11/04/12	9
11	Revisión documentación y pruebas código mensajes entre pc y mote	2 días	jue 12/04/12	vie 13/04/12	10
12	Caso de uso CSENS01: código lectura sensores por nodo de control	5 días	sáb 14/04/12	mié 18/04/12	11
13	Caso de uso CCOM01: código envío comandos a nodo actuador	5 días	jue 19/04/12	lun 23/04/12	12
14	Integración código desarrollado (nodos de control y actuador)	10 días?	sáb 14/04/12	lun 23/04/12	
15	Redacción informe (PEC 2)	6 días	jue 19/04/12	mar 24/04/12	

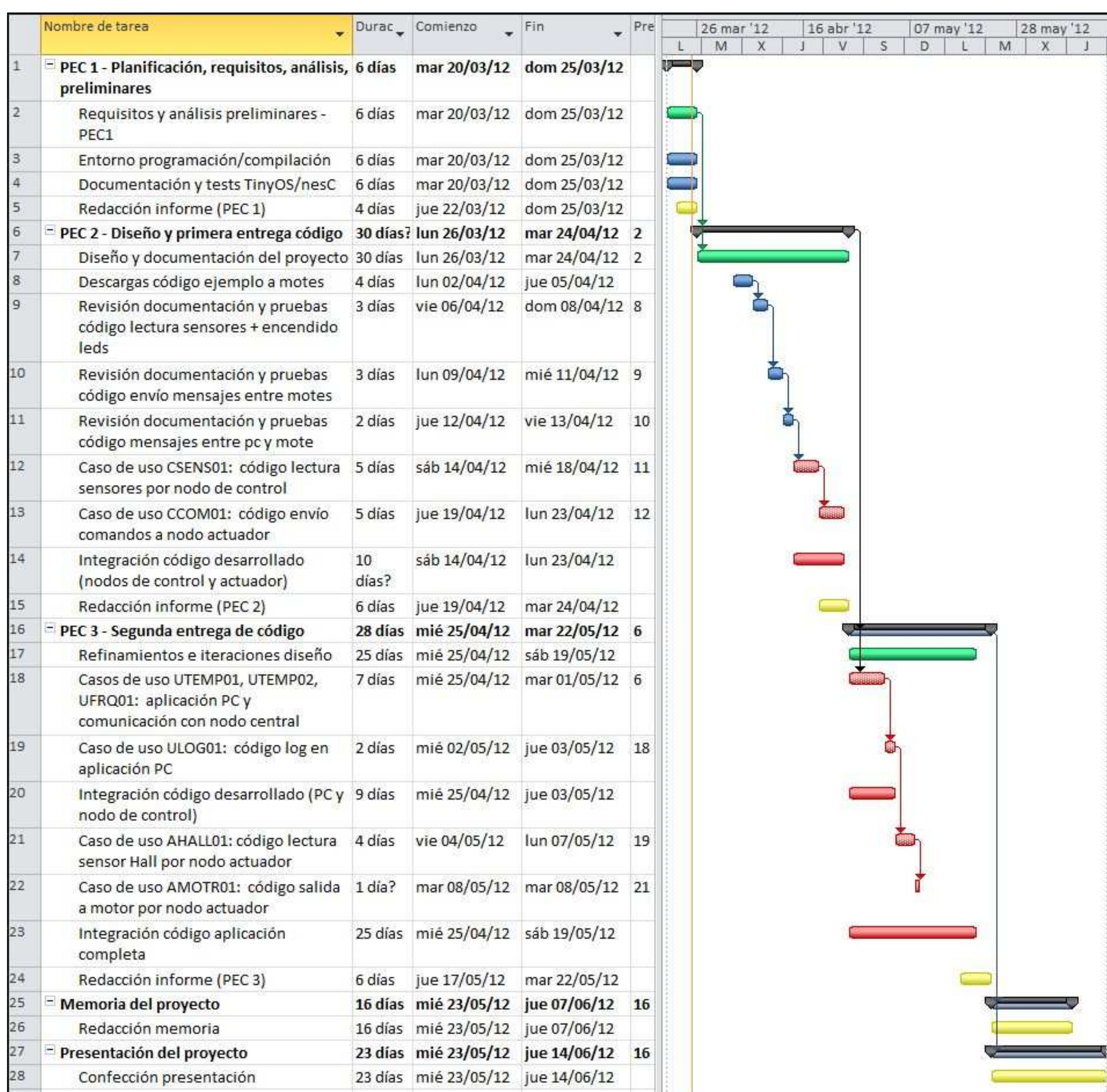
Por su parte, para la PEC 3 las tareas previstas y su planificación fueron:

	Nombre de tarea	Durac	Comienzo	Fin	Predece
16	**PEC 3 - Segunda entrega de código**	**28 días**	**mié 25/04/12**	**mar 22/05/12**	**6**
17	Refinamientos e iteraciones diseño	25 días	mié 25/04/12	sáb 19/05/12	
18	Casos de uso UTEMP01, UTEMP02, UFRQ01: aplicación PC y comunicación con nodo central	7 días	mié 25/04/12	mar 01/05/12	6
19	Caso de uso ULOG01: código log en aplicación PC	2 días	mié 02/05/12	jue 03/05/12	18
20	Integración código desarrollado (PC y nodo de control)	9 días	mié 25/04/12	jue 03/05/12	
21	Caso de uso AHALL01: código lectura sensor Hall por nodo actuador	4 días	vie 04/05/12	lun 07/05/12	19
22	Caso de uso AMOTR01: código salida a motor por nodo actuador	1 día?	mar 08/05/12	mar 08/05/12	21
23	Integración código aplicación completa	25 días	mié 25/04/12	sáb 19/05/12	
24	Redacción informe (PEC 3)	6 días	jue 17/05/12	mar 22/05/12	

Finalmente, las presentaciones tanto de la memoria como la final del proyecto, quedaron encajadas temporalmente de la siguiente manera:

	Nombre de tarea	Durac	Comienzo	Fin	Predece
25	**Memoria del proyecto**	**16 días**	**mié 23/05/12**	**jue 07/06/12**	**16**
26	Redacción memoria	16 días	mié 23/05/12	jue 07/06/12	
27	**Presentación del proyecto**	**23 días**	**mié 23/05/12**	**jue 14/06/12**	**16**
28	Confección presentación	23 días	mié 23/05/12	jue 14/06/12	

El resumen completo de la planificación inicial del proyecto se encuentra en el siguiente diagrama de Gantt, en el que se codifican en verde las tareas de desarrollo previas a la implementación (requisitos, análisis y diseño), en rojo las de implementación y programación, en amarillo las administrativas, y en azul las que podríamos denominar de infraestructura:



	Nombre de tarea	Durac	Comienzo	Fin	Pre
1	**PEC 1 - Planificación, requisitos, análisis, preliminares**	**6 días**	**mar 20/03/12**	**dom 25/03/12**	
2	Requisitos y análisis preliminares - PEC1	6 días	mar 20/03/12	dom 25/03/12	
3	Entorno programación/compilación	6 días	mar 20/03/12	dom 25/03/12	
4	Documentación y tests TinyOS/nesC	6 días	mar 20/03/12	dom 25/03/12	
5	Redacción informe (PEC 1)	4 días	jue 22/03/12	dom 25/03/12	
6	**PEC 2 - Diseño y primera entrega código**	**30 días?**	**lun 26/03/12**	**mar 24/04/12**	**2**
7	Diseño y documentación del proyecto	30 días	lun 26/03/12	mar 24/04/12	2
8	Descargas código ejemplo a motes	4 días	lun 02/04/12	jue 05/04/12	
9	Revisión documentación y pruebas código lectura sensores + encendido leds	3 días	vie 06/04/12	dom 08/04/12	8
10	Revisión documentación y pruebas código envío mensajes entre motes	3 días	lun 09/04/12	mié 11/04/12	9
11	Revisión documentación y pruebas código mensajes entre pc y mote	2 días	jue 12/04/12	vie 13/04/12	10
12	Caso de uso CSENS01: código lectura sensores por nodo de control	5 días	sáb 14/04/12	mié 18/04/12	11
13	Caso de uso CCOM01: código envío comandos a nodo actuador	5 días	jue 19/04/12	lun 23/04/12	12
14	Integración código desarrollado (nodos de control y actuador)	10 días?	sáb 14/04/12	lun 23/04/12	
15	Redacción informe (PEC 2)	6 días	jue 19/04/12	mar 24/04/12	
16	**PEC 3 - Segunda entrega de código**	**28 días**	**mié 25/04/12**	**mar 22/05/12**	**6**
17	Refinamientos e iteraciones diseño	25 días	mié 25/04/12	sáb 19/05/12	
18	Casos de uso UTEMP01, UTEMP02, UFRQ01: aplicación PC y comunicación con nodo central	7 días	mié 25/04/12	mar 01/05/12	6
19	Caso de uso ULOG01: código log en aplicación PC	2 días	mié 02/05/12	jue 03/05/12	18
20	Integración código desarrollado (PC y nodo de control)	9 días	mié 25/04/12	jue 03/05/12	
21	Caso de uso AHALL01: código lectura sensor Hall por nodo actuador	4 días	vie 04/05/12	lun 07/05/12	19
22	Caso de uso AMOTR01: código salida a motor por nodo actuador	1 día?	mar 08/05/12	mar 08/05/12	21
23	Integración código aplicación completa	25 días	mié 25/04/12	sáb 19/05/12	
24	Redacción informe (PEC 3)	6 días	jue 17/05/12	mar 22/05/12	
25	**Memoria del proyecto**	**16 días**	**mié 23/05/12**	**jue 07/06/12**	**16**
26	Redacción memoria	16 días	mié 23/05/12	jue 07/06/12	
27	**Presentación del proyecto**	**23 días**	**mié 23/05/12**	**jue 14/06/12**	**16**
28	Confección presentación	23 días	mié 23/05/12	jue 14/06/12	

5.2 La planificación definitiva:

En general, las variaciones sobre la planificación inicial no han sido importantes, aunque para la PEC 3 es de reseñar que se han tomado en consideración las variaciones en los casos de uso que han ido surgiendo conforme se refinaban los modelos, variaciones que han originado algunas modificaciones en la planificación: aparece alguna tarea nueva (correspondiente a la gestión de los casos nuevos), y se ha analizado el posible coste en tiempo de las modificaciones, lo que ha supuesto algunas variaciones en la duración de las tareas.

Tras decidir posponer al final el trabajo sobre la aplicación PC (que puede suponer una mejor ordenación de las tareas de programación al quedar continuas en el tiempo las que se refieren a programación bajo TinyOS), también se han intercambiado las fechas de implementación de los correspondientes casos de uso. No obstante, los resultados finales no han supuesto desviaciones de trascendencia en ningún concepto.

La planificación definitiva para el hito de entrega de la PEC 3 quedó, tras los cambios comentados, de la siguiente manera:

	Nombre de tarea	Durac	Comienzo	Fin	Predece
16	**PEC 3 - Segunda entrega de código**	**28 días**	**mié 25/04/12**	**mar 22/05/12**	**6**
17	Refinamientos e iteraciones diseño	25 días	mié 25/04/12	sáb 19/05/12	15
18	Casos de uso AHALL01y CHALL01: código lectura sensor Hall por nodo actuador y transmisión a control	4 días	mié 25/04/12	sáb 28/04/12	15
19	Caso de uso AMOTR01: código salida a motor por nodo actuador	2 días	dom 29/04/12	lun 30/04/12	18
20	Integración código desarrollado (nodos de control y actuador)	6 días	mié 25/04/12	lun 30/04/12	
21	Casos de uso UPARAM01 y CPARAM01: aplicación PC y comunicación con nodo de control	4 días	mar 01/05/12	vie 04/05/12	19
22	Caso de uso CLOG01: actualizar log [actúa en nodo de control y PC]	6 días	sáb 05/05/12	jue 10/05/12	21
23	Caso de uso ULOG01: acceso log en aplicación PC	2 días	vie 11/05/12	sáb 12/05/12	22
24	Integración código desarrollado (PC y nodo de control)	12 días	mar 01/05/12	sáb 12/05/12	
25	Integración código aplicación completa	25 días	mié 25/04/12	sáb 19/05/12	
26	Redacción informe (PEC 3)	6 días	jue 17/05/12	mar 22/05/12	

La planificación para la realización de la memoria y de la presentación del proyecto no ha sufrido variación alguna.

Por otra parte, al final del proyecto, y para una mejor presentación de la aplicación, se ha realizado como tarea adicional el desarrollo de una interface gráfica de usuario, característica que no estaba contemplada de forma explícita inicialmente. No obstante, su rápido desarrollo no ha modificado las líneas de tiempo básicas, y se ha implementado sin incidencias ni retrasos en otras tareas dentro de los periodos contemplados para la integración de los diferentes códigos de la aplicación completa.

6 - CASOS DE USO DE LA APLICACIÓN

Las descripciones informales de la aplicación que hemos visto en los apartados anteriores nos van a servir de fundamento para la captura y representación de una manera más formal de los requisitos y del comportamiento deseado del sistema.

Sabemos que un caso de uso recoge la descripción de las acciones que pueden ser ejecutadas por un actor sobre una aplicación para obtener los resultados deseados por el diseñador de la misma. Por actor nos referimos, habitualmente, a un rol desempeñado por una persona, un dispositivo, u otra aplicación al interactuar con el sistema. Empezaremos por identificar y analizar a nuestros actores.

6.1 Los actores y sus funciones en la aplicación:

De los análisis básicos y preliminares de los requisitos y del diseño hardware, podemos distinguir los siguientes actores:

- **Usuario de la aplicación:** persona que de acuerdo a su juicio *parametriza* el sistema para fijar los umbrales de temperatura que activarán las acciones del nodo actuador, así como la frecuencia de lectura de los sensores y de envío de comandos desde el nodo de control al actuador. Además, revisa el log del sistema para analizar su funcionamiento.

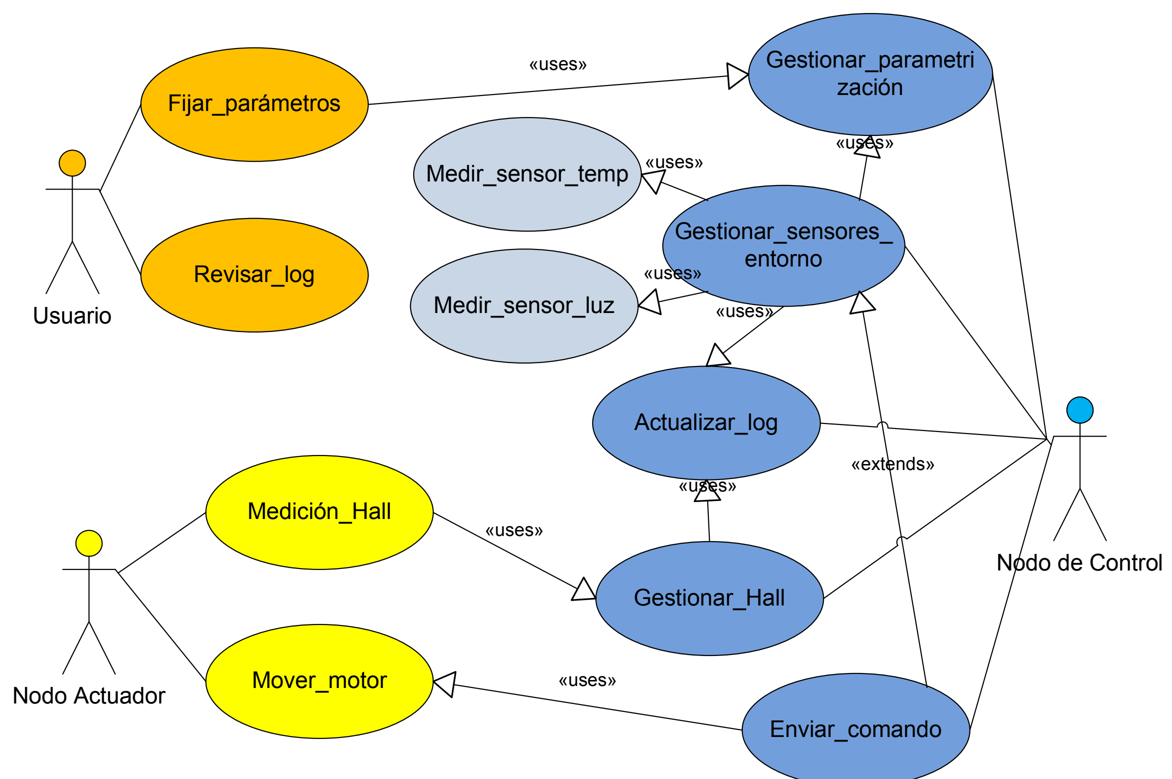
- **Nodo de control (o central):** sistema implementado en una de las *mote* que, de acuerdo a una parametrización previa, recibe y trata las medidas de los sensores para posteriormente, y de acuerdo a una programación adecuada, enviar a través de una comunicación inalámbrica comandos de actuación al nodo actuador. Por otra parte, registra las medidas de los sensores cuando se producen, además de las posibles acciones tomadas de acuerdo a esas medidas, para lo que transmite la correspondiente información a una aplicación PC.

- **Nodo actuador**: sistema implementado en una segunda *mote* que recibe los comandos del nodo de control y los ejecuta con la finalidad de gobernar los dos posibles movimientos de un motor acoplado al mecanismo de lamas: abrir o cerrar. Además, y mediante un sensor de efecto Hall que queda enfrentado al campo magnético de un imán cuando las lamas están completamente cerradas, es capaz de detectar esa situación particular de la persiana y transmitirla para su registro al nodo central.

6.2 Casos de uso:

Como es usual en las diferentes metodologías, los productos que vamos obteniendo en cualquiera de las fases del desarrollo del software están sujetos a posteriores refinamientos conforme se avanza y profundiza en los fundamentos de la aplicación o sistema a desarrollar. En general, y salvo que se diga lo contrario, ofrecemos en este documento las últimas versiones de los productos de cada fase de desarrollo, que supondrán la culminación de las diferentes iteraciones que se han podido efectuar sobre cada resultado.

De acuerdo a lo anterior, presentamos los casos de uso finalmente identificados y calificados bajo la descripción gráfica que nos aporta un diagrama de casos de uso bajo UML:



Aclaramos y precisamos a continuación, y para cada uno de los actores identificados, estos casos de uso en un formato estructurado, que incluye descripciones informales de los mismos pero que intentaremos sean lo más completas posible.

6.2.1 Actor Usuario:

Para el actor ***Usuario***, tendremos:

<table>
<tr><td colspan="3">Nombre del caso de uso: Fijar_parámetros</td><td>Código: UPARAM01</td></tr>
<tr><td colspan="4">Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012</td></tr>
<tr><td>Actor:</td><td>Usuario</td><td>Frecuencia:</td><td>Discrecional</td></tr>
<tr><td>Descripción:</td><td colspan="3">El usuario fija los parámetros de la aplicación (temperaturas superior, inferior y frecuencia de muestreo), que serán transmitidos al nodo de control</td></tr>
<tr><td>Precondiciones:</td><td colspan="3"></td></tr>
<tr><td>Postcondiciones:</td><td colspan="3">El sistema queda parametrizado con el nuevo umbral introducido</td></tr>
<tr><td>Pseudocódigo flujo normal</td><td colspan="3">1. Usuario introduce los correspondientes parámetros en la aplicación PC
2. Aplicación PC transmite vía serie los nuevos parámetros al nodo de control
3. Aplicación PC registra en el log los nuevos parámetros
4. El nodo de control modifica la parametrización en su lógica
5. El nodo de control ejecuta inmediatamente su lógica para tomar en consideración el nuevo umbral de temperatura máxima</td></tr>
<tr><td>Pseudocódigo flujos alternativos</td><td colspan="3">Si los parámetros introducidos no son válidos, entonces mensaje de error</td></tr>
<tr><td>Extiende</td><td colspan="3"></td></tr>
<tr><td>Incluye</td><td colspan="3">Gestionar_parametrización (actor Nodo de Control)</td></tr>
<tr><td>Generaliza</td><td colspan="3"></td></tr>
</table>

<table>
<tr><td colspan="3">Nombre del caso de uso: Revisar_log</td><td>Código: ULOG01</td></tr>
<tr><td colspan="4">Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012</td></tr>
<tr><td>Descripción:</td><td colspan="3">El usuario accede al contenido del fichero log del sistema para su revisión</td></tr>
<tr><td>Actor:</td><td>Usuario</td><td>Frecuencia:</td><td>Discrecional</td></tr>
<tr><td>Precondiciones:</td><td colspan="3">Existe un fichero log en el que se vuelcan medidas de los sensores y acciones del sistema</td></tr>
<tr><td>Postcondiciones:</td><td colspan="3"></td></tr>
<tr><td>Pseudocódigo flujo normal</td><td colspan="3">Usuario introduce el correspondiente comando y se visualiza el contenido del fichero de log</td></tr>
<tr><td>Pseudocódigo flujos alternativos</td><td colspan="3">Si el fichero no existe o no es accesible, entonces mensaje de error</td></tr>
<tr><td>Extiende</td><td colspan="3"></td></tr>
<tr><td>Incluye</td><td colspan="3"></td></tr>
<tr><td>Generaliza</td><td colspan="3"></td></tr>
</table>

6.2.2 Actor Nodo de Control:

<table>
<tr><td colspan="3">Nombre del caso de uso: Gestionar_parametrización</td><td>Código: CPARAM01</td></tr>
<tr><td colspan="4">Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012</td></tr>
<tr><td>Descripción:</td><td colspan="3">El nodo de control recibe del nodo de usuario los parámetros y los registra en la aplicación</td></tr>
<tr><td>Actor:</td><td>Nodo de control</td><td>Frecuencia:</td><td>Parámetro</td></tr>
<tr><td>Precondiciones:</td><td colspan="3">El usuario ha introducido en el sistema los nuevos parámetros</td></tr>
<tr><td>Postcondiciones:</td><td colspan="3">El sistema ha quedado parametrizado con los nuevos parámetros</td></tr>
<tr><td>Pseudocódigo flujo normal</td><td colspan="3">1. El nodo de control recibe mensaje de la aplicación de usuario con los nuevos parámetros
2. Los nuevos parámetros se fijan en las correspondientes variables</td></tr>
<tr><td>Pseudocódigo flujos alternativos</td><td colspan="3"></td></tr>
<tr><td>Extiende</td><td colspan="3"></td></tr>
<tr><td>Incluye</td><td colspan="3"></td></tr>
<tr><td>Generaliza</td><td colspan="3"></td></tr>
</table>

<table>
<tr><td colspan="3">Nombre del caso de uso: Gestionar_sensores_entorno</td><td>Código: CSENS01</td></tr>
<tr><td colspan="4">Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012</td></tr>
<tr><td>Descripción:</td><td colspan="3">El nodo de control toma muestras de los sensores y las trata para computar una actuación sobre el motor; inicia los procesos de enviar comando al nodo actuador y de actualizar el log tanto con las lecturas de los sensores como con los comandos decididos</td></tr>
<tr><td>Actor:</td><td>Nodo de control</td><td>Frecuencia:</td><td>Parámetro</td></tr>
<tr><td>Precondiciones:</td><td colspan="3">• Existe una frecuencia de toma de mediciones
• Existen sensores activos</td></tr>
<tr><td>Postcondiciones:</td><td colspan="3">• Las medida de las muestras alimentarán la lógica del sistema y se habrá creado, en su caso, un mensaje con la finalidad de mover la persiana
• Se actualizará el fichero log por la muestra y, en su caso, por el comando decidido</td></tr>
<tr><td>Pseudocódigo flujo normal</td><td colspan="3">1. El temporizador del nodo activa la función de medida de los sensores
2. El nodo de control toma muestras de los sensores
3. La medida se contrasta con los parámetros y decide actuación:
a) Si medición entre umbrales, no se actúa
b) Si medición fuera de umbrales, prepara comando (mensaje) para enviar al nodo actuador (ver caso de uso Enviar_comando)
4. La medición y el posible comando se graban en el fichero log mediante el correspondiente mensaje a la aplicación PC</td></tr>
<tr><td>Pseudocódigo flujos alternativos</td><td colspan="3">Si no puede tomarse muestra del sensor, entonces mensaje de error al log</td></tr>
<tr><td>Extiende</td><td colspan="3"></td></tr>
<tr><td>Incluye</td><td colspan="3">Medir_sensor_temp, Medir_sensor_luz, Actualizar_log</td></tr>
<tr><td>Generaliza</td><td colspan="3"></td></tr>
</table>

Vemos que este último caso incluye a los más particulares sobre la medición de los sensores de luz y temperatura. No los especificamos por

resultar obvios. Podría resolverse en dos casos: uno correspondiente a la medición propiamente dicha y otro para referenciar la toma de decisión de acuerdo a las mediciones. Dada la integración y correlación entre ambos, simplificamos haciendo explícito un único caso de uso.

<table>
<tr><td colspan="3">Nombre del caso de uso: Gestionar_Hall</td><td>Código: CHALL01</td></tr>
<tr><td colspan="4">Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012</td></tr>
<tr><td>Descripción:</td><td colspan="3">El nodo de control recibe notificación de un evento Hall desde el nodo actuador, y la comunica al log</td></tr>
<tr><td>Actor:</td><td>Nodo de control</td><td>Frecuencia:</td><td>Al cerrarse del todo la cortina, o bien al empezar a abrirse</td></tr>
<tr><td>Precondiciones:</td><td colspan="3">Se habrá registrado en el nodo actuador un evento Hall, lo que equivale a que la persiana se ha cerrado del todo o ha comenzado a abrirse.</td></tr>
<tr><td>Postcondiciones:</td><td colspan="3">Se actualizará el fichero log por el evento que ha tenido lugar</td></tr>
<tr><td>Pseudocódigo flujo normal</td><td colspan="3">1. El nodo de control recibe mensaje del nodo actuador representativo de un evento Hall
2. El nodo de control computa la nueva situación de la persiana
3. Se actualiza el log con la nueva situación de la persiana</td></tr>
<tr><td>Pseudocódigo flujos alternativos</td><td colspan="3"></td></tr>
<tr><td>Extiende</td><td colspan="3"></td></tr>
<tr><td>Incluye</td><td colspan="3">Actualizar_log</td></tr>
<tr><td>Generaliza</td><td colspan="3"></td></tr>
</table>

<table>
<tr><td colspan="3">Nombre del caso de uso: Enviar_comando</td><td>Código: CCOM01</td></tr>
<tr><td colspan="4">Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012</td></tr>
<tr><td>Descripción:</td><td colspan="3">El nodo de control envía un comando al nodo actuador como resultado de la gestión de las lecturas de los sensores</td></tr>
<tr><td>Actor:</td><td>Nodo de control</td><td>Frecuencia:</td><td>Igual o inferior al CSENS01, siempre que haya comando</td></tr>
<tr><td>Precondiciones:</td><td colspan="3">Existe una decisión de acción tomada por la lógica del sistema, representada en un comando a enviar al nodo actuador mediante un mensaje</td></tr>
<tr><td>Postcondiciones:</td><td colspan="3">• Se habrá transmitido un comando al nodo actuador, lo que equivale a que se habrá movido la persiana
• Se habrá actualizado el fichero log por la acción tomada y transmitida</td></tr>
<tr><td>Pseudocódigo flujo normal</td><td colspan="3">1. La aplicación del nodo de control envía mensaje con el comando correspondiente al nodo actuador
2. Se recibe confirmación de la ejecución del comando por el nodo actuador
3. Se actualiza el log por la acción ejecutada</td></tr>
<tr><td>Pseudocódigo flujos alternativos</td><td colspan="3">• Si no puede transmitirse el mensaje por el nodo central, entonces mensaje de error al log
• Si no se recibe confirmación de la ejecución de la acción por parte del nodo actuador, entonces mensaje de error al log</td></tr>
<tr><td>Extiende</td><td colspan="3">Gestionar_sensores_entorno</td></tr>
<tr><td>Incluye</td><td colspan="3">Mover_motor (nodo actuador)</td></tr>
<tr><td>Generaliza</td><td colspan="3"></td></tr>
</table>

Este caso de uso podría incluirse en el ya visto Gestionar_sensores_entorno. La complejidad que alcanzaría este último, así como las particularidades hardware de Enviar_comando, parecen aconsejar la diferenciación conceptual de estos dos casos.

<table>
<tr><td colspan="3">Nombre del caso de uso: Actualizar_log</td><td>Código: CLOG01</td></tr>
<tr><td colspan="4">Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012</td></tr>
<tr><td>Descripción:</td><td colspan="3">El nodo de control actualiza el log por las lecturas de los sensores, los comandos enviados, o los eventos Hall recibidos. Se desarrolla entre el nodo de control y el PC</td></tr>
<tr><td>Actor:</td><td>Nodo de control</td><td>Frecuencia:</td><td>Igual CSENS01, más evento Hall</td></tr>
<tr><td>Precondiciones:</td><td colspan="3">Se han tomado medidas de los sensores, y ha actuado la lógica de la aplicación para decidir, o no, un comando; o bien ha ocurrido un evento Hall</td></tr>
<tr><td>Postcondiciones:</td><td colspan="3">• Se habrá transmitido un mensaje desde el nodo de control al PC del usuario
• Se habrá actualizado el fichero log por los motivos del mensaje</td></tr>
<tr><td>Pseudocódigo flujo normal</td><td colspan="3">1. La aplicación del nodo de control envía mensaje con el contenido correspondiente al PC del usuario
2. Se actualiza el log según el contenido del mensaje</td></tr>
<tr><td>Pseudocódigo flujos alternativos</td><td colspan="3"></td></tr>
<tr><td>Extiende</td><td colspan="3"></td></tr>
<tr><td>Incluye</td><td colspan="3"></td></tr>
<tr><td>Generaliza</td><td colspan="3"></td></tr>
</table>

6.2.3 Actor Nodo Actuador:

Nombre del caso de uso: Mover_motor			**Código:** AMOTR01
Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012			
Descripción:	El nodo actuador activa el motor tras recibir comando del nodo de control		
Actor:	Nodo actuador	**Frecuencia:**	Igual al CCOM01
Precondiciones:	Existe decisión de acción tomada por la lógica del sistema, representada en un comando/mensaje enviado al, y recibido por, el nodo actuador		
Postcondiciones:	• Se habrá transmitido una señal al motor, lo que implica que se habrá realizado un movimiento (abrir/cerrar) en la cortina • Se habrá enviado un mensaje de confirmación de ejecución de la acción al nodo de control		
Pseudocódigo flujo normal	1. El nodo actuador recibe un mensaje del nodo central conteniendo un comando 2. El nodo actuador envía una señal al motor para que se active durante un periodo determinado 3. Se confirma la ejecución de la acción al nodo central		
Pseudocódigo flujos alternativos			
Extiende			
Incluye			
Generaliza			

Nombre del caso de uso: Medición_Hall			**Código:** AHALL01
Autor: Agustín Isasa | UOC | PFC Sistemas Empotrados | 2º sem. 2011-2012			
Descripción:	El nodo actuador recibe activación sensor Hall y lo notifica al nodo de control		
Actor:	Nodo actuador	**Frecuencia:**	Al cerrarse la cortina, o al empezar a abrirse
Precondiciones:	El sensor de efecto Hall del nodo actuador se encuentra activo y su salida conectada, vía posible interrupción, al microcontrolador del nodo		
Postcondiciones:	• El movimiento de las lamas ha originado variaciones en la salida del sensor Hall, bien porque se han encontrado el sensor Hall y el campo magnético (imán) situado apropiadamente en el mecanismo de la persiana, bien porque se han separado • Se habrá enviado un mensaje de confirmación del evento al nodo de control • El *log* queda actualizado por la inclusión del evento		
Pseudocódigo flujo normal	1. El nodo actuador detecta variación en salida del sensor Hall 2. El nodo transmite un mensaje representativo del evento al nodo central, incluyendo la nueva medición 3. El nodo central analiza la medición y actualiza el log por el evento de cierre (o no cierre) de las lamas		
Pseudocódigo flujos alternativos			
Extiende			
Incluye			
Generaliza			

7 - DISEÑO

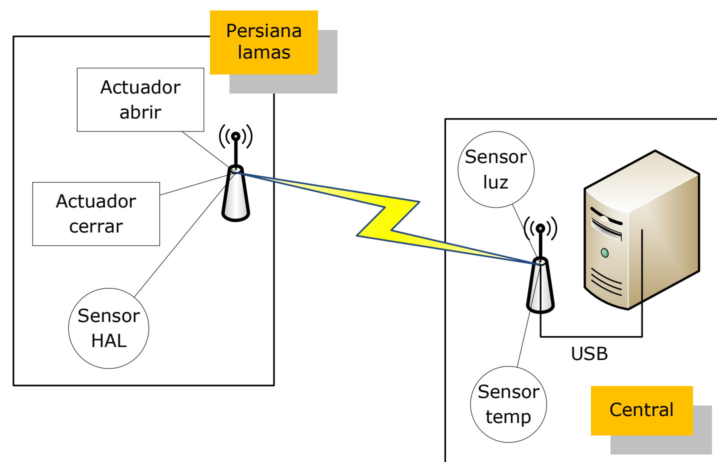
7.1 Punto de partida para el diseño:

En relación al proceso de desarrollo de software, es conveniente recordar los principales productos obtenidos con carácter previo al inicio de las tareas de diseño de la aplicación:

1. El modelo informal de la aplicación que obteníamos a partir de la descripción textual del dominio y de las necesidades que pretendemos resolver con el sistema que nos ocupa (ver el capítulo *2 – La aplicación a desarrollar*).

2. Los requisitos de la aplicación que representábamos más formalmente con la enumeración de los casos de uso y con las descripciones textuales de sus fundamentos (ver el capítulo *6 – Casos de uso de la aplicación*).

Resumíamos el primer punto con el siguiente diagrama de despliegue de hardware:



Por su parte, y en relación a los casos de uso, en el apartado correspondiente los organizábamos en torno a los tres actores identificados (Usuario, Nodo de control y Nodo actuador), tratamos de abstraer cada caso de uso representativo en la aplicación completa, relacionamos de manera estructurada (aunque con representación textual de sus principales características) cada caso de uso, y representamos su resumen mediante el diagrama UML de casos de uso de la página 28.

7.2 El proceso de desarrollo:

Hemos aplicado, en general, un proceso de desarrollo del software acomodado en lo necesario al *Proceso Unificado de Desarrollo* de Jacobson, Booch y Rumbaugh. Como sabemos, este proceso de desarrollo se caracteriza por estar dirigido por los casos de uso, estar centrado en la arquitectura, y resultar iterativo e incremental. Además, está basado en componentes, lo que equivale a decir que el sistema software se descompone en componentes, también de naturaleza software, interrelacionados a través de interfaces bien definidas.

No obstante, tanto el proceso de desarrollo como la representación de sus productos a través de UML (*Lenguaje Unificado de Modelado*), son lo suficientemente flexibles como para modular su utilización según las necesidades y circunstancias de cada proyecto concreto. En particular, y dadas las características de nuestro sistema en construcción, creímos que en esta fase del desarrollo resultaría adecuada la representación de un modelo situado entre los de diseño e implementación que nos permitiera:

- Reflexionar sobre los principales conceptos que los casos de uso (y por lo tanto nuestro sistema) manejan: su abstracción mediante clases de diseño nos permitirá analizar tanto sus características (atributos) como las acciones que podremos ejecutar sobre las mismas (métodos), aunque siendo conscientes de que la orientación a objetos en este contexto es más una ayuda conceptual que un fin del diseño/implementación en sí mismo.
- Vincular estos (seudo)objetos a los componentes que los implementan: por definición, cuando tratamos de componentes estamos tratando ya de potencial código que contará con correspondencia directa con módulos software futuros y reales a desarrollar.
- Vincular los componentes a los nodos que los acogen: nuestro sistema está distribuido en tres nodos, por lo que esta abstracción estará, de nuevo, en correspondencia directa con la realidad.
- De acuerdo a todo lo anterior, acelerar la transición entre diseño e implementación y, por tanto, el inicio de la construcción del código.

Como vemos, la primera característica, según la cual trabajamos clases (probablemente de diseño), está canónicamente vinculada a la fase de diseño del Proceso Unificado de Desarrollo. No así los modelos de componentes o de despliegue que se apuntan en las características posteriores, y que corresponden ya a la fase de implementación. Pero, como se ha avanzado, la interrelación entre estos modelos para representar un sistema no excesivamente complejo como el que nos ocupa nos permitió una visión global más adecuada.

Representaremos a continuación el diseño inicial de nuestra aplicación bajo los anteriores criterios para, más adelante, representar la adaptación de estos esquemas a lo que resultaron los componentes reales bajo TinyOS. En realidad no realizamos una traducción o adaptación directas, sino que consideramos el modelo TinyOS como un refinamiento de los anteriores diseños (y en particular del que se va a presentar a continuación), y resultado, en definitiva, de una nueva iteración sobre estos modelos previos.

7.3 Diseño de la aplicación:

Es necesario recordar aquí que los casos de uso, como la mayor parte de los productos del proceso de desarrollo de software, pueden ser objeto de variaciones de acuerdo a las reflexiones que sobre ellos pueden realizarse en cada iteración del proceso; es por ello que los primeros diagramas que vamos a representar pueden contener algunas divergencias con respecto al modelo de casos de uso que hemos presentado en el capítulo anterior ya que, para repetir, se trataba del último modelo de los realizados.

Actuando con los casos de uso como referente para la dirección de nuestro diseño, identificamos abstracciones de diseño para cada uno de los actores. En nuestro caso, además, resultó práctico vincular cada actor con uno de los nodos hardware del sistema, si bien esta relación no es absolutamente exacta en términos lógicos (el nodo `PC Usuario`, por ejemplo, no sólo implementa casos de uso del actor Usuario, sino que también implementa parcialmente casos de los otros actores).

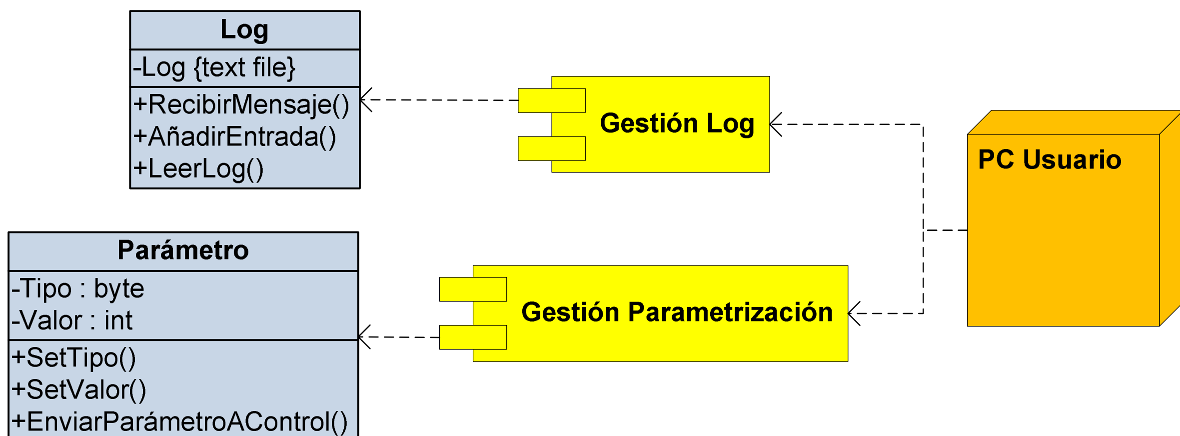
En los siguientes diagramas, y de izquierda a derecha, efectuamos una transición desde la visión que nos proporcionan las clases de diseño (que para nuestros efectos, y como se ha anticipado, no tendrán una vinculación directa con objetos de la aplicación, sino que nos permitirán una revisión conceptual del dominio) hasta la realidad física de los nodos de nuestro sistema, pasando por la abstracción de los componentes software que deberían transformarse de forma directa en código real (aunque sujetos, de acuerdo al proceso y como veremos, a refinamientos posteriores). Cada elemento del diagrama se vincula mediante relaciones de dependencia con otros elementos de acuerdo a la siguiente lógica:

- Los componentes software establecen una relación de dependencia con las clases que implementan.
- Los nodos establecen una relación de dependencia con los componentes que despliegan.
- A este nivel no se explicitan las relaciones o interfaces entre componentes. Estas quedan implícitas, pero ocultas de momento,

en las *cajas negras* que constituyen los nodos, pero se pondrán de manifiesto cuando refinemos el modelo y lo adaptemos a TinyoOS.

7.3.1 Diseño para el actor Usuario:

De acuerdo a lo anterior, y para el actor *Usuario* (y el nodo `PC Usuario`), podemos establecer el siguiente diseño:

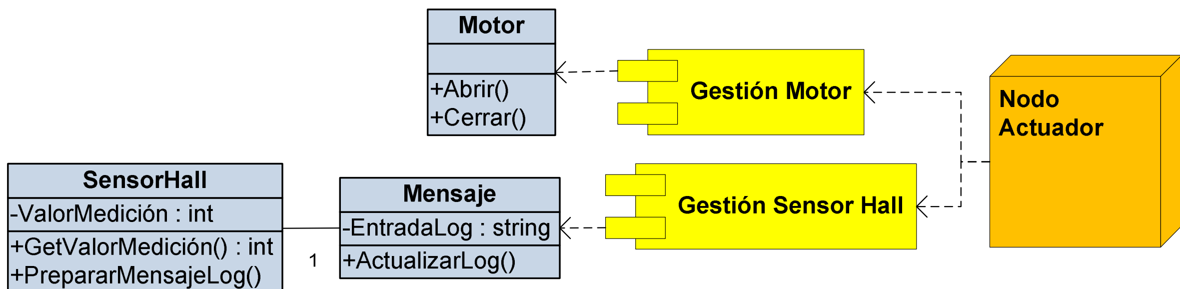


Apreciamos dos componentes relacionados directamente con los casos de uso del actor. El componente `Gestión Log` implementa el caso *Revisar_log*, además de ocuparse también de porciones de otros casos de uso de otros actores (como la actualización del log por los casos de uso que hacen referencia al registro de las lecturas de los sensores o a los comandos ordenados). La clase de diseño `Log`, que nos permite refinar el principal objeto del componente, nos apunta su implementación física mediante un fichero de texto, así como las funciones vinculadas a añadir una entrada en el log tras su recepción, así como a la posible revisión por parte del usuario. Como veremos, estos métodos darán lugar a las correspondientes interfaces.

Por su parte, el componente `Gestión Parametrización` implementa el caso de uso identificado sobre el actor Usuario que hace referencia a la fijación de los parámetros del sistema: temperatura superior, inferior, y frecuencia de muestreo. La clase `Parámetro` abstrae la realidad de este objeto: un determinado tipo (uno de los tres que hemos visto), su valor (de acuerdo al juicio del usuario), y los métodos para asignar valores a estos atributos, así como su envío al nodo de control mediante el correspondiente método.

7.3.2 Diseño para el actor Nodo Actuador:

Para el actor Nodo *Actuador*, y para el nodo de igual nombre, podemos representar:



Identificamos tres objetos principales para ser trabajados en este nodo: `Motor`, que abstrae el elemento físico que actúa sobre las persianas; `SensorHall`, que representa al sensor de efecto Hall de nuestro sistema; y `Mensaje`, un atributo adicional del sensor cuya relevancia preferimos hacer explícita representándolo mediante su propia clase.

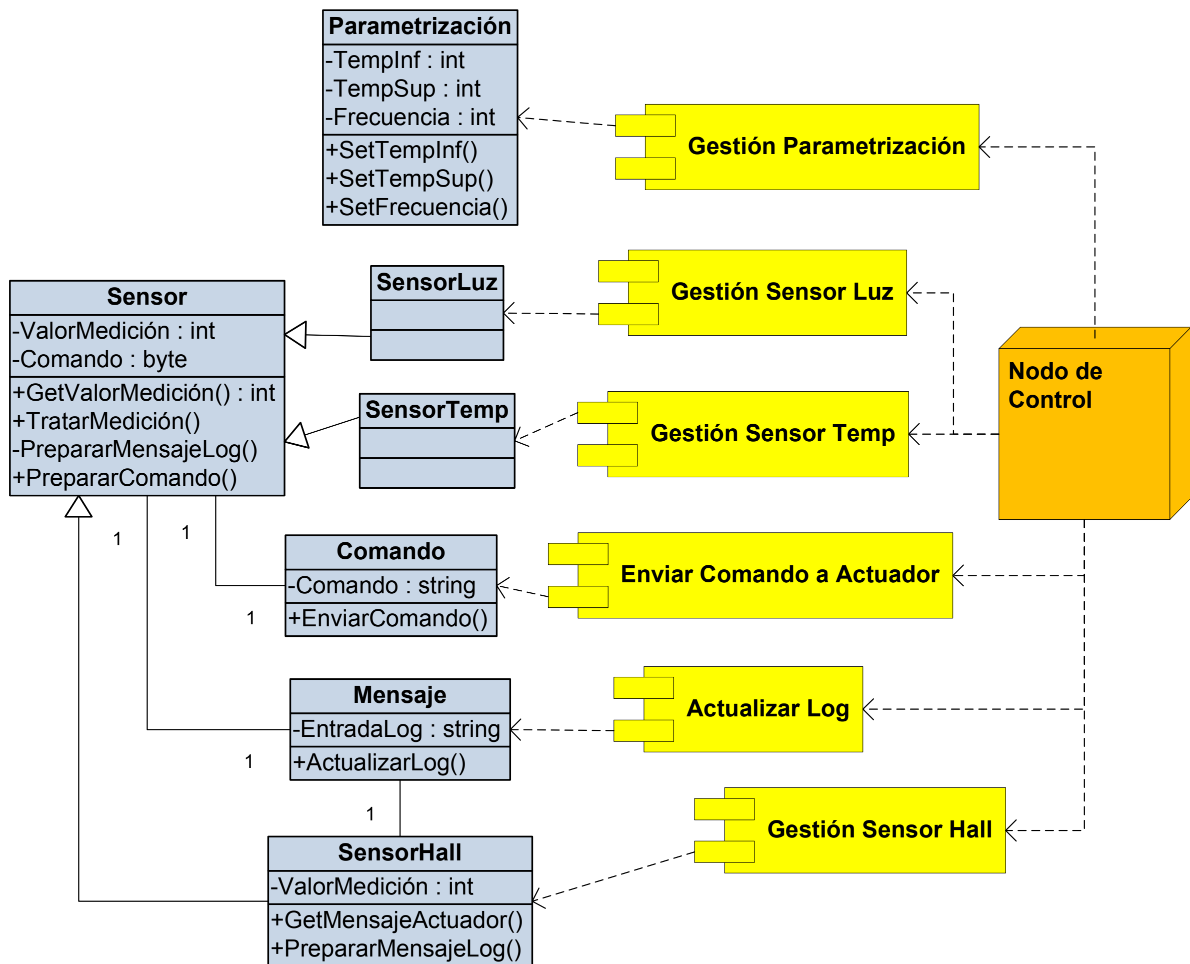
Siguiendo bajo la dirección de los casos de uso, identificamos dos componentes estratégicos y directamente vinculados a los dos casos de uso inicialmente expuestos, y que este nodo-actor debe desplegar: el que hace referencia a la gestión del motor (caso de uso *Mover_motor*) y el que hace referencia a la gestión del sensor Hall (caso de uso *Medición_Hall*).

7.3.3 Diseño para el actor Nodo de Control:

Por último, para el actor *Nodo de Control* vamos a ver una mayor complejidad en el diseño puesto que este nodo implementa la principal lógica de la aplicación, y es en buena parte intermediario entre el hardware y software distribuidos y la aplicación (y nodo) del usuario.

En este caso veremos cómo del refinamiento de los casos de uso (probablemente por la iteración efectuada tras la reflexión sobre los objetos -conceptuales-, sus atributos, y las acciones sobre estos que deberán implementar los componentes) identificamos y pusimos de manifiesto alguno más de los inicialmente previstos. En este último sentido también tiene que ver la *transversalidad* entre nodos de los casos de uso y por tanto de la aplicación distribuida, y por la que cabía prever un posible desglose de algunos casos de uso entre cada nodo partícipe de su realización, y que permitiera recoger una abstracción más cercana a la implementación.

Tenemos, por tanto:



De entre los conceptos que pretendemos poner de manifiesto representándolos mediante las clases de diseño, cabe destacar los relativos a los sensores. De la clase general `Sensor` se identifican especializaciones para el sensor de luminosidad, el de temperatura y el de efecto Hall. No obstante, el tratamiento sólo es idéntico para los de temperatura y luminosidad (muestreo, tratamiento del valor medido, elaboración si procede del comando, y registro en log), mientras que el de Hall es una especialización que sólo requiere el despacho de la información recibida desde el nodo Actuador (recepción del mensaje y registro en log). Precisar que el método `PrepararMensajeLog()` de esta clase `Sensor` puede ser una parte de `TratarMedición()`, o una función auxiliar y privada de esta última, lo que se decidirá en una nueva iteración sobre el diseño/implementación en el momento de la codificación.

Como atributos adicionales a `Sensor`, se identifican las clases `Comando` y `Mensaje`, que se hacen explícitas dada la conveniencia de los objetos que representan para transmitir el diseño del sistema.

Para terminar con las clases, `Parametrización` abstrae el conjunto de parámetros que debe manejar la lógica de la aplicación para sus fines.

Aparte de las salvedades vistas, en el resto de los casos no parece necesaria mayor explicación sobre atributos o métodos.

En relación a los componentes de este subsistema o nodo, recordaremos que habíamos identificado dos casos de uso principales: el que hace referencia al envío de comandos al nodo actuador y el que se relaciona con la gestión (lectura, proceso y toma de decisiones) de los sensores de temperatura y luz del entorno. Vinculados directamente con estos casos de uso, tenemos los siguientes componentes:

- `Gestión Sensor Luz` y `Gestión Sensor Temp`: casos de uso *Medir_sensor_luz* y *Medir_sensor_temp*, respectivamente, y que como veremos algo más adelante se fusionaron en un único componente `Gestión Sensores Entorno`.
- `Enviar Comando a Actuador`: caso de uso *Enviar_comando*.

Además, en el nodo que nos ocupa habrá que desplegar los siguientes componentes adicionales:

- `Gestión Parametrización`: es una proyección del caso de uso del actor Usuario correspondiente a la parametrización del sistema, y mediante la cual se reciben los parámetros del usuario y se registran en la lógica.
- `Actualizar Log`: es una proyección de los casos de uso del propio Nodo de Control relativos a la medición de los sensores (incluyendo la transmisión de los valores del sensor Hall). Reseñar que se desarrolla tanto en el nodo de control como en el PC.
- `Gestión Sensor Hall`: proyección del caso de uso *Medición_Hall* del Nodo Actuador en el Nodo de Control para reenviar su resultado al log.

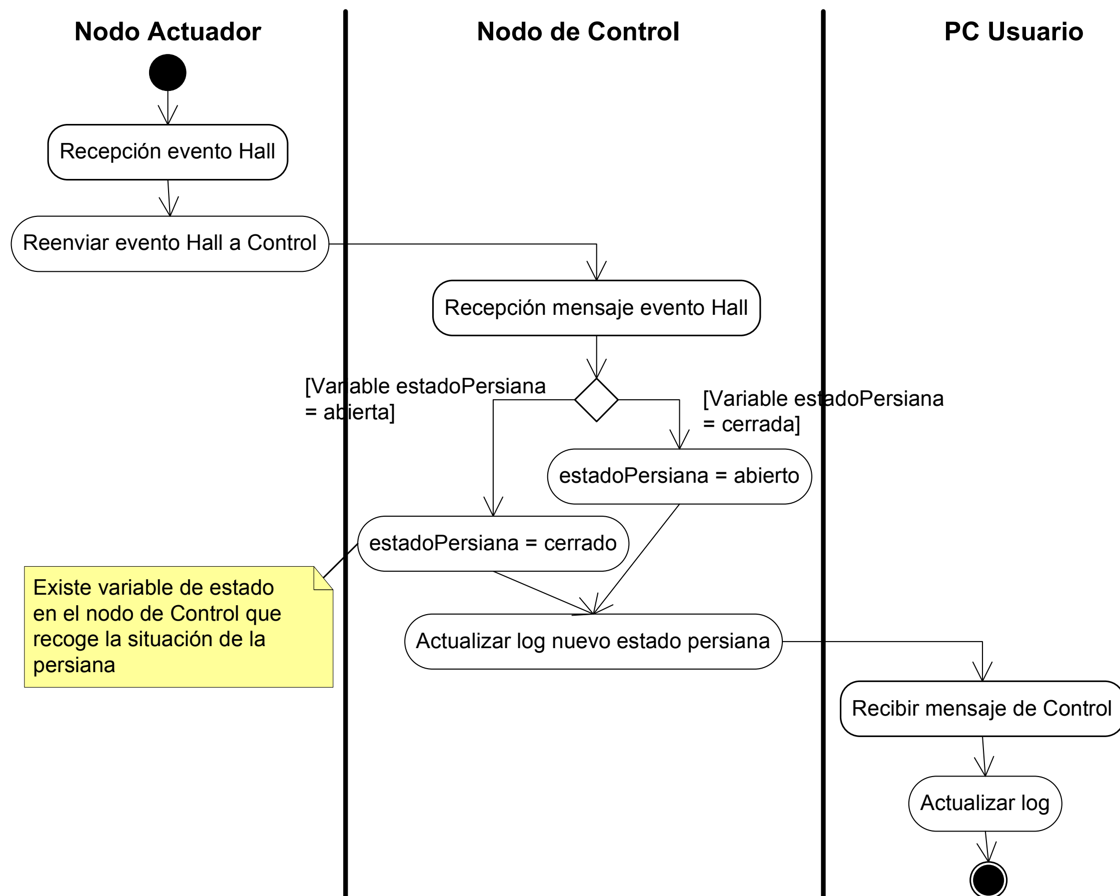
A nuestros efectos, y por simplicidad, nos resultó conveniente una relación biunívoca entre casos de uso y componentes software, lo que nos proporciona un menor nivel de abstracción y nos acerca cada vez más a la implementación. Podríamos especificar que estamos ante lo que en UML se define como una *realización de caso de uso-diseño*, pero no debemos olvidar que esta absoluta identificación de casos de uso en componentes es un artefacto conveniente, y que podría haberse resuelto asignando diversos componentes a un único caso de uso, o viceversa.

7.4 Detalle de la lógica de la aplicación:

Antes de trabajar la adaptación del diseño anterior (y en particular los componentes UML) a TinyOS, conviene especificar de forma adecuada la lógica de la aplicación, cuyo núcleo hasta este momento ha venido considerándose implícito de manera incompleta y totalmente informal en todos los informes previos. La especificación de esta lógica, además, nos sirvió para seguir refinando los modelos de diseño e implementación, así como su aproximación al modelo final de TinyOS y, posteriormente, a la implementación.

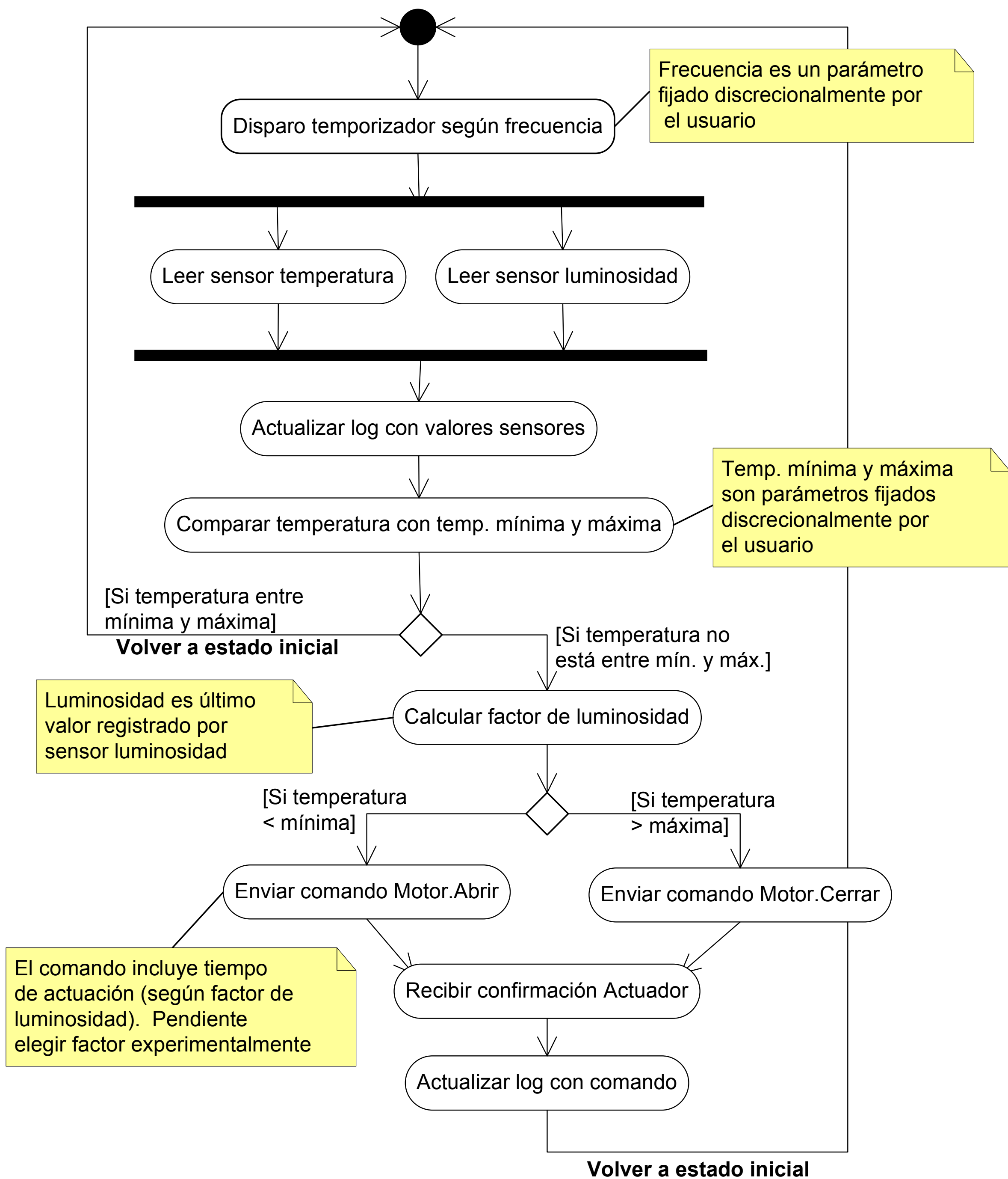
Sabemos que la comparación de las lecturas de los sensores de temperatura y luminosidad contra unos parámetros previamente fijados por el usuario podrá originar órdenes para transferir señales a un motor que moverá, en un sentido u otro, una persiana. Además, un sensor de efecto Hall permitirá saber si la persiana está completamente cerrada o no. Todo lo anterior quedará reflejado en un log.

Veamos con cierto detalle en el siguiente diagrama UML de actividades las tareas que se desencadenan para el caso de uso *Medición_Hall*, es decir, cuando el sensor de efecto Hall situado en el nodo actuador registra una variación en el campo magnético (la persiana se ha cerrado del todo, o bien se ha empezado a abrir; dadas las características del sensor no hay posibilidad de una mayor información o precisión en la captura del estado):



Como vemos, el caso de uso se desarrolla a lo largo de los tres nodos: se inicia con el evento Hall en el nodo actuador, se comunica el evento vía radio al nodo de control, y finalmente se registra en el log implementado en el PC del usuario a través de una comunicación serie entre el nodo de control y el PC.

Por lo que se refiere al núcleo de la aplicación, es decir, el control de la apertura o cierre de la persiana en función de la temperatura y luminosidad, consideraremos que si la temperatura leída está fuera de un rango predeterminado se efectuará un último contraste contra un factor de luminosidad que tomará en consideración el valor leído del sensor fotoeléctrico y un factor o constante a determinar experimentalmente, y de acuerdo al resultado se decidirá actuar (o no) sobre el motor, el sentido de esta actuación (para abrir o cerrar la persiana), y su tiempo de duración. Tendremos la siguiente representación del bucle de actividades, en la que sólo nos ocupamos de las tareas que se desarrollan en el nodo de control:



Cabría realizar otros diagramas para otras actividades (como, por ejemplo, la parametrización del sistema, la actualización del log, la interrelación entre los diferentes nodos de acuerdo a la lógica principal...), pero son tareas no excesivamente complejas para las que es suficiente, como hasta ahora, su descripción textual. Tampoco se han incluido flujos alternativos o control de errores, que suponen en su caso refinamientos de posteriores iteraciones.

7.5 Adaptación del diseño a TinyOS:

Una vez revisada la arquitectura y descomposición en componentes (al menos los más básicos) de TinyOS, y pertrechados con el modelo de casos de uso refinado, el diagrama básico de clases y componentes (diseño e implementación), y con la especificación más formal de las principales funcionalidades del sistema, pudimos pasar a adaptar el diseño al entorno de TinyOS, lo que nos situó definitivamente en el ámbito de la implementación.

Puesto que el despliegue de la aplicación TinyOS se efectúa de forma distribuida en dos *motes* o nodos, y puesto que cada nodo tendrá su propia, y diferente del otro, imagen, tendremos dos diagramas de componentes, uno para cada *mote*. Obviamos en este apartado la aplicación PC y su interrelación con el nodo de control, para la que se han sentado sus bases al ocuparnos de la arquitectura cliente-servidor entre TinyOS y PC en el apartado sobre la arquitectura de TinyOS.

Seguiremos utilizando UML y aprovechándonos de su flexibilidad, e intentaremos hacer explícita la diferenciación entre los subsistemas propios del sistema operativo y los que implementarán la lógica de la aplicación.

Empezamos por la conceptualmente más sencilla aplicación del nodo actuador.

7.5.1 Nodo Actuador:

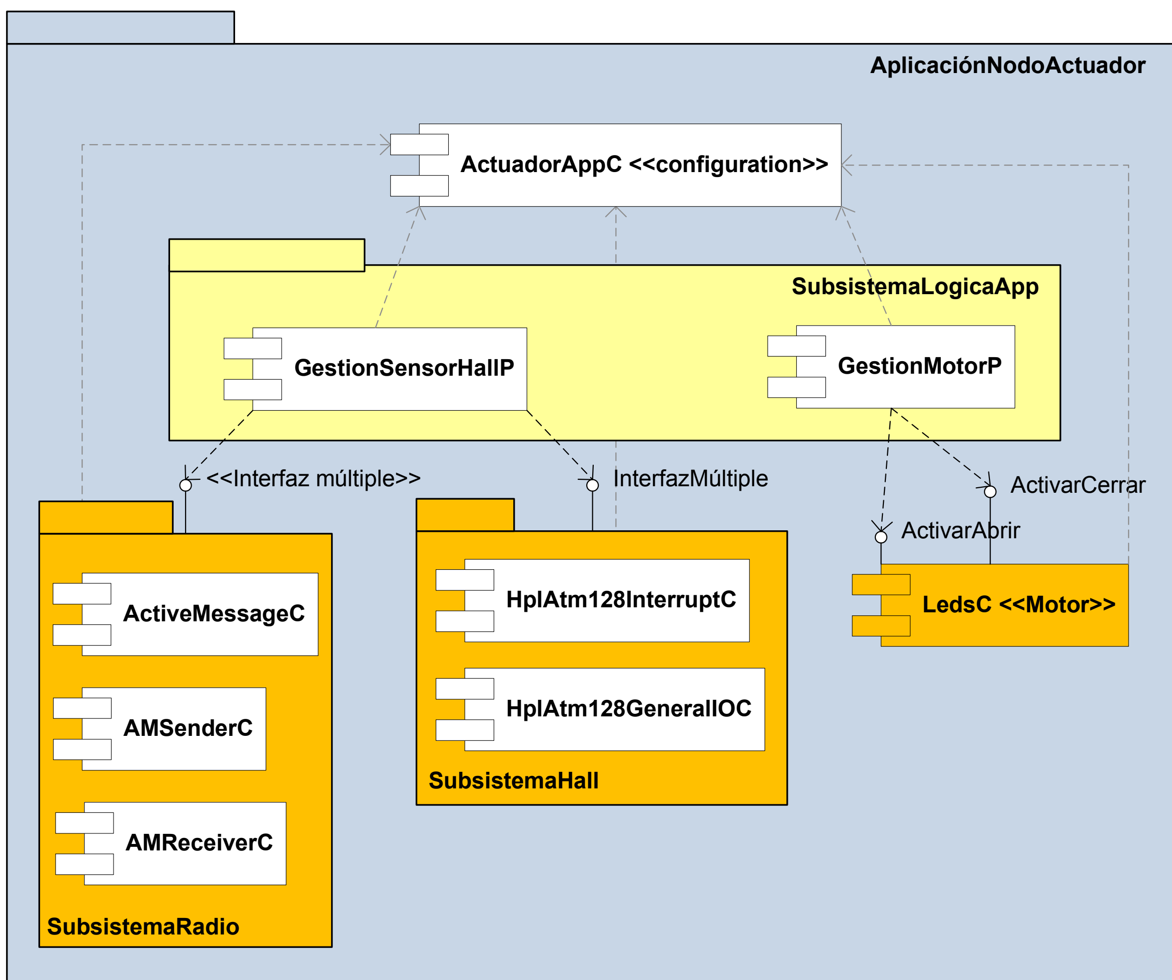
Identificamos tres subsistemas auxiliares del sistema operativo que vamos a necesitar:

- Con el subsistema Radio podremos recibir en este nodo los comandos ordenados por el nodo de control. Además, podremos enviar por esta vía la notificación del disparo del sensor de efecto Hall.
- Con la denominación de subsistema Hall identificamos al conjunto de componentes que nos van a permitir detectar y gestionar la interrupción que originará el sensor Hall cuando se someta a un campo magnético.
- También utilizaremos el componente de TinyOS que soporta la gestión de los leds del nodo. Recordemos que utilizaremos los leds para representar las señales enviadas al motor (abrir y cerrar la persiana).
- Hacemos explícito el subsistema que implementa la lógica de la aplicación en este nodo, y cuyos componentes se relacionan

directamente a los identificados en el diagrama que ya disponíamos sobre clases de diseño y componentes.

- Las interfaces entre los componentes del sistema y de la lógica de la aplicación las estereotipamos mediante el indicador de interfaz múltiple, y que en la implementación tendrá que ser desarrollada en sus diferentes elementos.
- Por último, y como puede apreciarse, se establecen vinculaciones de dependencia de cada uno de los componentes de la aplicación con un nuevo componente de configuración que en la implementación resultará necesario de acuerdo al modelo de programación de TinyOS bajo nesC.

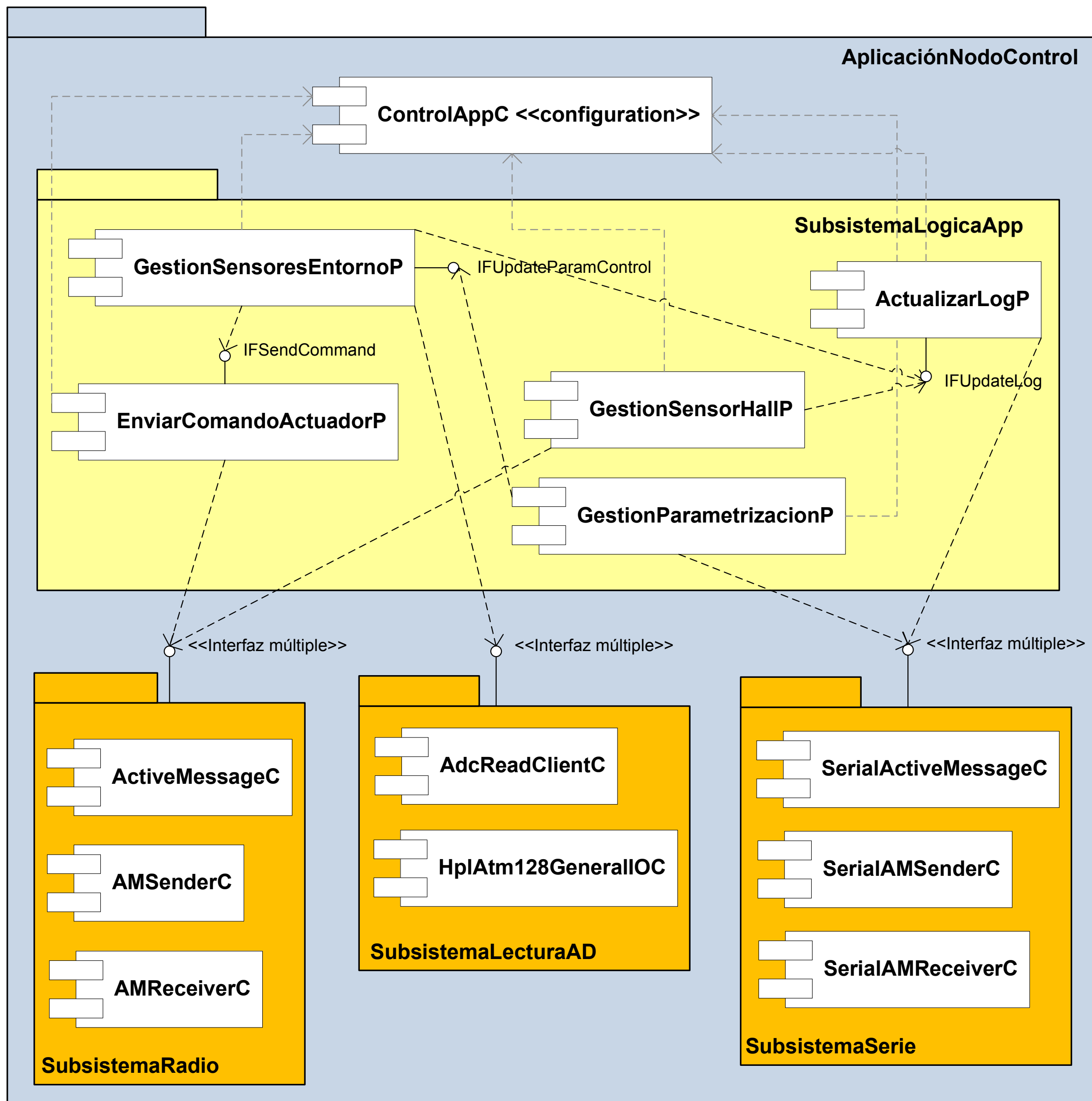
Tenemos, pues:



7.5.2 Nodo de Control:

Veamos ahora la algo más compleja funcionalidad de la aplicación que debe instalarse en el nodo de control.

De forma semejante a lo que acabamos de ver, tendremos un conjunto de subsistemas formados cada uno con un conjunto de componentes del sistema operativo, así como otro conjunto de componentes que implementará la lógica de la aplicación a desplegar en este nodo. El modelo resultante es:



Ponemos de manifiesto las novedades en este diagrama:

- Aparece un nuevo subsistema que agrupa los componentes necesarios para efectuar las lecturas analógicas de los sensores de temperatura y luminosidad. Este subsistema es utilizado por el componente que implementa la lógica de la aplicación que gestiona estas lecturas.

- También aparece el nuevo subsistema que se encarga de la comunicación serie entre el nodo y el PC, y que vemos resulta, en cuanto a su estructura en componentes, equivalente al subsistema de comunicación por radiofrecuencia. También veremos en su momento que es equivalente en términos de interfaces.
- Los componentes que implementan la lógica de la aplicación se relacionan directamente con los obtenidos en el diagrama inicial de diseño/implementación, con el matiz de que, como refinamiento en esta iteración, fusionamos en un único componente `GestionSensoresEntornoP` los dos que distinguíamos previamente para gestionar cada uno de los sensores de temperatura y luminosidad. Este componente único tendrá que recoger la lógica que se aplique a esas lecturas para decidir si se aplica, o no, una actuación sobre el motor y en qué sentido (si se abre la persiana o se cierra). Por tanto, ya no nos referimos a esta tarea como *medir sensores*, sino como *gestionar sensores del entorno*.
- Se identifican y especifican las interfaces necesarias entre los componentes de la lógica de la aplicación, mientras que para los componentes del sistema operativo mantenemos el estereotipo de interfaz múltiple.
- Para no recargar el diagrama omitimos las relaciones de los subsistemas del SO con respecto al componente de configuración.
- También se omite por simplicidad el subsistema de gestión de leds que resulta indicativo del funcionamiento del sistema, no fundamental en la funcionalidad de la aplicación pero sí muy útil, y que se implementó en la realización definitiva.

7.6 La interface de usuario:

Recordemos que al actor Usuario le correspondían dos casos de uso: *Fijar Parámetros* (código UPARAM01) y *Revisar Log* (código ULOG01). Por lo tanto, las funcionalidades básicas de la interface de usuario deberían ser:

1. Recibir del usuario mediante las entradas adecuadas los parámetros del sistema y transmitirlos al nodo de control.

2. Presentar al usuario, a solicitud de este, el contenido del log.

3. Como funcionalidad auxiliar, vinculada a la anterior, modificar el programa visor del fichero de texto que constituye el log.

Estas funcionalidades básicas pueden resolverse con un sencillo diseño de la interface como el que se presenta en los siguientes prototipos para presentación en consola o bajo escritorio gráfico:

```
----------------------------------------
C O N T R O L   P E R S I A N A   M O V I L
----------------------------------------

          == MENU PRINCIPAL ==

PARAMETROS ACTUALES:

    Temperatura mínima:  21
    Temperatura máxima:  25
    Frecuencia muestreo: 15

    Visor del log: gedit

OPCIONES DEL PROGRAMA:
    1 - Revisar log
    2 - Modificar parámetros
    3 - Cambiar editor para revisar log

> Introduzca opción [1, 2 o 3]:
```

Como se aprecia, la ventana de la aplicación se divide en dos partes: en la superior se reflejan los valores de la parametrización actual del sistema, mientras que en la inferior se presenta el menú principal, en el que podrá pulsarse el botón 'Revisar log' para que este se abra en el correspondiente visor (además de poder cambiar este último programa), así como modificar los parámetros del sistema.

Se toma la decisión de ofrecer las dos interfaces presentadas: la gráfica y la de consola. Esta última puede resultar útil para hardware con pocos recursos o para sistemas que no cuenten con escritorio gráfico. Ambas interfaces se intentan acomodar en lo posible a una semejante distribución de las funcionalidades que deben servir. Pueden verse más pantalla de la interface bajo consola en el manual del usuario (Anexo 2).

8 - IMPLEMENTACIÓN

8.1 Nodo de control:

8.1.1 `GestionSensoresEntornoP` y `EnviarComandoActuadorP`:

Los primeros casos de uso a implementar según la planificación elaborada, correspondían al muestreo y gestión de los sensores (código CSENS01) y al envío de comandos al nodo actuador (CCOM01), ambos del actor Nodo de Control.

En términos de componentes se trataba de la codificación de `GestionSensoresEntornoP` y `EnviarComandoActuadorP`. Recordaremos que la funcionalidad del primero incluye tanto la lectura de los sensores de temperatura y luz como la gestión de la lógica de la aplicación por lo que se refiere a la toma de decisiones para actuar, o no, sobre el motor de la persiana. En cuanto al segundo, transmitirá al nodo actuador los comandos que el componente anterior pueda decidir.

Además, la evidente interrelación entre ambos hizo necesaria la implementación de una interface a una función que tendrá que ser llamada por `GestionSensoresEntornoP` cuando quiera enviar un comando al nodo actuador a través de `EnviarComandoActuadorP`. A esta interface la denominamos `IFSendCommand`, y se implementa en el componente `EnviarComandoActuadorP` añadiéndole la función `sendCmd()` según exige la declaración de la interface. A esta función se le pasa como parámetro un entero definido de acuerdo a su semántica en el fichero de cabecera `ControlApp.h`, y que representará el comando transmitido. También se le pasa otro entero con el tiempo de actuación sobre el motor. En ese mismo fichero *header* se especifica el formato del mensaje que encapsulará el comando, y del que hace uso `EnviarComandoActuadorP` para su transmisión vía radio al nodo actuador.

La declaración de la interface obligará, además, a la implementación de un *handler* en el componente que llame a la función `IFSendCommand`, y mediante el cual se manejará el evento `sendCmdDone(error_t err)`, que acaece cuando se ha transmitido un comando al nodo actuador vía radio.

Para el desarrollo del código correspondiente a la lectura de los sensores hemos utilizado como fundamentos los de la aplicación de ejemplo *blinkCou*, adaptándolo en lo necesario para nuestros fines. Esta aplicación, tras las correspondientes inicializaciones del hardware, maneja una iteración permanente entre diferentes estados que leen los sensores y vuelcan las lecturas obtenidas mediante comunicación serie.

En el *Anexo 1: Una nota sobre el cálculo de la temperatura ambiente*, explicamos de forma más detallada el sencillo cálculo de conversión de las lecturas del sensor a temperatura ambiente.

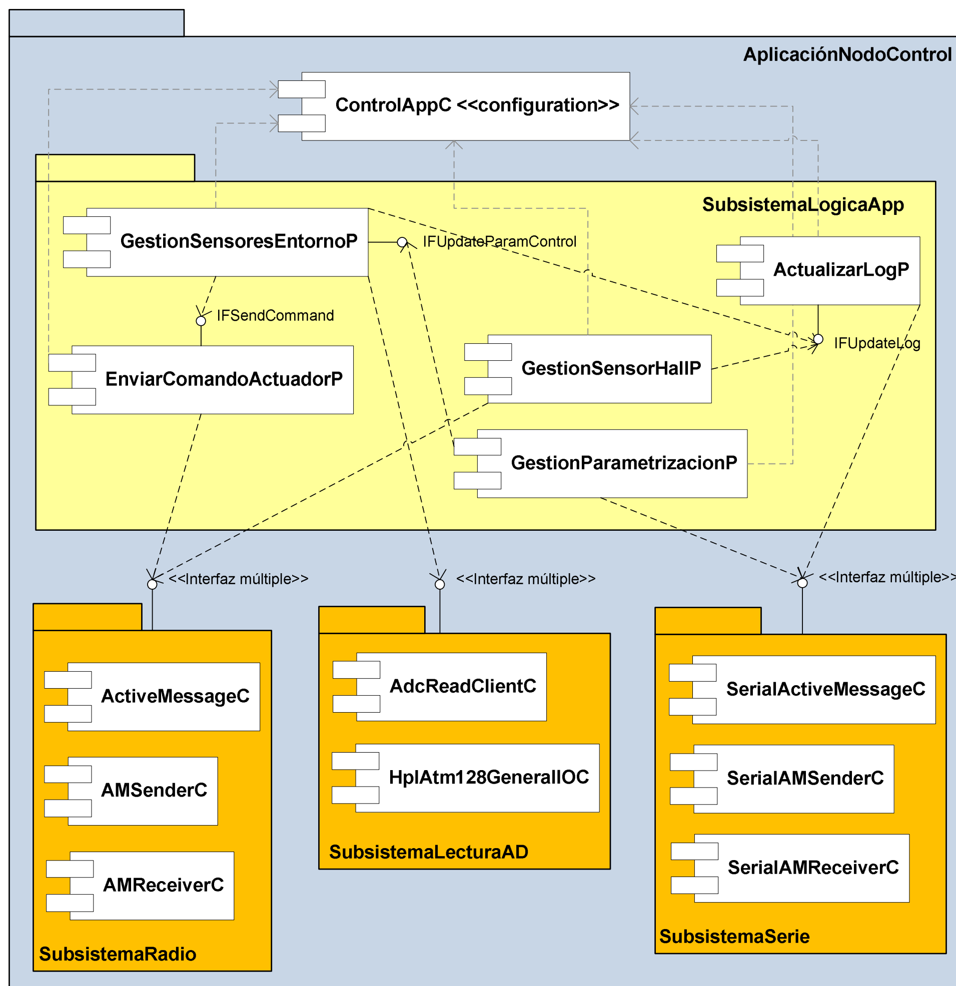
En las posteriores iteraciones para la construcción del código completo, ha habido que realizar diferentes modificaciones, adiciones y ajustes sobre el código de estos primeros casos de uso. Las principales han correspondido a:

- Se delega al componente `EnviarComandoActuadorP` la inicialización de la radio. Resolvemos añadir a la interface `IFSendCommand` una función adicional `radioOn()` cuya llamada habilitará el uso de la radio en la aplicación del nodo. De forma simétrica, la interface se completa con el manejador `radioOnDone(error_t err)`, que gestiona el evento que se produce cuando la radio queda inicializada correctamente.

- Por otra parte, y tras diferentes observaciones experimentales, se le da forma prácticamente definitiva a la lógica de decisión y actuación basada en las medidas de los sensores, fijando los factores de las diferentes fórmulas empleadas.

- El componente `GestionSensoresEntornoP` ha tenido que ser completado con el código correspondiente a su interacción con el componente `ActualizarLogP`, que implementa el caso de uso de igual nombre (CLOG01). Veremos esta interacción con más detalle cuando nos refiramos al código de este componente algo más adelante.

Si recordamos el modelo de diseño/implementación para el nodo de control, vemos cómo encajaba esta pareja de componentes en la visión completa de la aplicación para este nodo (en el siguiente diagrama seguimos omitiendo, para mayor claridad, la dependencia de los diversos componentes con el subsistema de gestión de los leds y con el marco de ejecución de la aplicación):



Con este último modelo como referencia, veamos ahora el resto de componentes del nodo desarrollados.

8.1.2 `ActualizarLogP`:

Vinculado directamente con el caso de uso de nombre equivalente (código CLOG01), se trata, como podemos apreciar en el modelo que acabamos de ver, de un componente que recibe mensajes de otros componentes que retransmitirá, vía puerto serie, hasta el log implementado en la aplicación de usuario que se ejecuta en un PC.

Implementa una función a la que se le pasan como parámetros un puntero al contenido (*payload*) de un mensaje y su longitud, discrimina de acuerdo a esta última qué tipo de mensaje es, y lo transmite al log mediante la función estándar de comunicación serie `AMSend.send(…)`.

Para conseguir la funcionalidad buscada, `ActualizarLogP` debe proporcionar la interface `IFUpdateLog`, en la que encontramos la función principal, ya apuntada, `sendToLog(void* payload, uint8_t len)`. La

interface se complementa con un manejador del evento representativo de la actualización del log, `logDone(error_t err)`.

Además, y de forma equivalente a lo que vimos en el apartado anterior para la habilitación del subsistema de radio, se delega en este componente la habilitación del subsistema de comunicaciones vía puerto serie. La interface `IFUpdateLog` se completa finalmente con la función `serieOn()` y el *handler* `serieOnDone(error_t err)`.

El componente `ActualizarLogP` es utilizado por `GestionSensoresEntornoP` para transmitir al log los mensajes con las lecturas de los sensores y los comandos transmitidos al motor como resultado de esas lecturas. También lo usa `GestionSensorHallP` para transmitir al log las entradas correspondientes a eventos del sensor de efecto Hall detectados por el nodo actuador y transmitidos para su gestión al nodo de control.

8.1.3 `GestionSensorHallP`:

A este componente (caso de uso *Gestionar Hall* - CHALL01), como hemos ido viendo en el proceso de diseño, le encomendamos la recepción de la comunicación de disparos del sensor de efecto Hall situado en el nodo actuador (eventos Hall en lo sucesivo), y que acaecen cuando la persiana se cierra completamente, o cuando se empieza a abrir.

Se articula en torno a la función estándar de recepción de mensajes radio `Receive.receive(message_t* msg, void* payload, uint8_t len)`, de la que se extrae el mensaje enviado por el nodo actuador vía radio. Este mensaje es genérico, y se encomienda al nodo de control (y al componente que nos ocupa en particular) su gestión para decidir si la persiana se acaba de cerrar o se está empezando a abrir (para ello utiliza la numeración del evento Hall, y conociendo la situación inicial de la persiana y, por tanto, el valor inicial por defecto, conocerá para números impares del evento que la persiana acaba de cerrarse, y lo contrario para numeraciones pares; con ello logramos, además, evitar la pérdida de sincronización que supondría una pérdida de un mensaje y nos evitamos una rutina adicional de comprobación, ya que la información se extrae de su numeración).

Una vez recibido el mensaje del actuador y averiguada la situación de la persiana, hacemos uso de la interface `IFUpdateLog` y de su función `sendToLog(…)`, que hemos visto en el apartado anterior, para transmitir al log el evento Hall y la situación actual de la persiana.

8.1.4 `GestionParametrizacionP`:

El último componente del nodo de control que nos queda por revisar está directamente vinculado con el caso de uso de igual nombre (código CPARAM01), y tiene como función la recepción de los parámetros del sistema (rango de temperaturas mínima y máxima, y frecuencia de muestreo -y, por tanto, de decisión de acciones- de los sensores) desde la aplicación de usuario que corre en el PC, y su transmisión al componente GestionSensoresEntornoP.

Para ello, el código de este componente incluye la función estándar de recepción serie `receive(message_t* msg, void* payload, uint8_t len)`, de la interface `Receive`, implementada en el componente de la distribución TinyOS `SerialAMReceiverC`, y de la que recibe los parámetros fijados por el usuario desde su aplicación que corre en el PC.

Una vez recibidos, transmite los tres parámetros del sistema a `GestionSensoresEntornoP`, para lo que este último componente proporciona la interface `IFUpdateParamControl`. Esta interface declara la función `setParameters(uint8_t tInf, uint8_t tSup, uint16_t freqOp)`, así como el manejador `setParamDone(error_t err)`, a implementar por el módulo que use la interface. Las firmas de la función y del *handler* se pretende sean autoexplicativas.

8.1.5 Otros elementos del código:

En los apartados previos hemos visto la implementación de los cinco módulos o componentes que conforman la aplicación del nodo de control, a los que hay que añadir tres ficheros adicionales que incluyen las declaraciones (funciones y eventos) de cada una de las tres interfaces vistas. Todos ellos son ficheros nesC con sufijo `.nc`.

Además de estos ocho ficheros, el código de esta aplicación incluye:

- Un fichero de configuración de la aplicación, `ControlAppC.nc`, exigido por nesC y que declara los componentes de la aplicación y el *cableado* (*wiring*) entre las interfaces de los componentes.
- Un fichero de cabecera (*header*), `ControlApp.h`, con las definiciones de las constantes compartidas y las estructuras bajo las que se representan los cuatro diferentes tipos de mensajes que nuestro sistema maneja (informativos con lecturas de sensores y comandos decididos, comandos, representación de eventos Hall, y parámetros).
- Un fichero `Makefile` en el que se comparten instrucciones de compilación tanto de la aplicación nesC del nodo de control como de la aplicación de usuario bajo Java. En resumen, para la

compilación de la aplicación nesC se exige dependencia de las clases Java, para cuya construcción se pasan las oportunas instrucciones (ver apartado correspondiente al código de la aplicación de usuario, así como el capítulo sobre el manual de usuario, donde se precisa la estructura, y sus motivos, de los ficheros del código fuente).

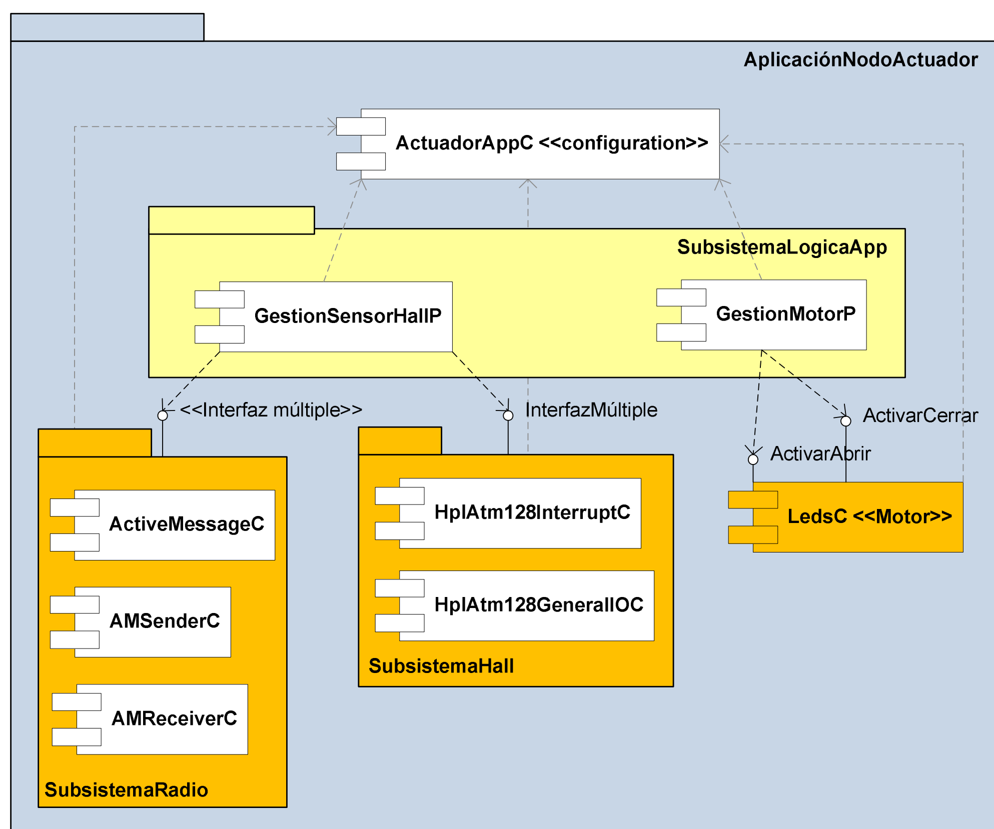
8.2 Nodo actuador:

Recordemos que corresponden al nodo actuador dos funciones básicas:

- Gestionar (capturar y reenviar al nodo de control) los cambios de voltaje en la salida del sensor de efecto Hall. Corresponde al caso de uso *Medición Hall* (AHALL01).

- Gestionar (recibir y aplicar al correspondiente circuito de salida) los comandos de actuación sobre el motor que transmite vía radio el nodo de control. Corresponde al caso de uso *Mover motor* (AMOTR01).

De manera directa, y de acuerdo a lo anterior, se identificaron en su momento dos componentes, `GestionSensorHallP` y `GestionMotorP`, que veíamos en el anterior informe cómo encajaban en la visión completa de la aplicación para el nodo actuador:



8.2.1 `GestionSensorHallP`:

Como hemos visto, implementa el caso de uso *Medición Hall* (AHALL01) del actor Nodo Actuador, y constituye la entrada de la aplicación al implementar el evento `Boot.booted()`, que inicializa los circuitos necesarios para activar las lecturas del sensor de efecto Hall e inicializa la radio.

En cuanto a funcionalidades del sistema, ya hemos visto que gestiona las lecturas del sensor de efecto Hall, que se detectan vía interrupción que ataca uno de los pins del microcontrolador. La rutina de servicio a esta interrupción, `HallPinInterrupt.fired()`, se limita a llamar a la tarea `comControl()`, desde la que se prepara el correspondiente mensaje y se envía al nodo de control por medio de la función estándar de transmisión por radio `AMSend.send(…)`.

8.2.2 `GestionMotorP`:

Implementa el caso de uso *Mover Motor* (código AMOTR01), para lo que incluye la función de recepción de mensajes radio `Receive.receive(…)`, que gestiona los mensajes incluyendo comandos (actuación sobre el motor y por cuánto tiempo) que envía el nodo de control a este nodo actuador. En función de qué movimiento hay que transmitir al motor (abrir la persiana o cerrarla) habilita una de dos salidas posibles del microcontrolador, que en nuestro prototipo derivamos a sendos leds.

Además, incorpora un temporizador que se dispara una vez transcurrido el periodo de actuación que se recibe en el mensaje, y a cuyo disparo cesa la señal de salida.

8.2.3 Otros elementos del código:

Además de estos dos ficheros que acabamos de ver, el código de esta aplicación incluye:

- Un fichero de configuración de la aplicación, `ActuadorAppC.nc`, exigido por nesC y que declara los componentes de la aplicación y el *cableado* (*wiring*) entre las interfaces de los componentes.
- Un fichero de cabecera (*header*), `ActuadorApp.h`, con las definiciones de las constantes compartidas y las estructuras bajo las que se representan los dos diferentes tipos de mensajes que la aplicación del nodo actuador maneja (comandos y representación de eventos Hall).
- Un fichero `Makefile` básico con las instrucciones de compilación de la aplicación nesC del nodo actuador.

8.3 Aplicación de usuario:

La aplicación de usuario, que debe ejecutarse en un PC al que se conecta vía USB (aunque envolviendo una comunicación serie RS232) el nodo de control, se resuelve con Java mediante una única aplicación de consola, para la que necesitaremos una clase principal y otras auxiliares que nos permitan instanciar los mensajes que recibe o emite.

Como venimos indicando, al actor Usuario le corresponden dos casos de uso: *Fijar Parámetros* (código UPARAM01) y *Revisar Log* (ULOG01). Además, sobre esta aplicación se proyecta el caso de uso *Actualizar Log* (CLOG01) del actor Nodo de Control. Por lo tanto, las funcionalidades básicas de la aplicación de usuario son:

1. Recibir del usuario mediante la interface adecuada los parámetros del sistema y transmitirlos al nodo de control. Esta nueva fijación de parámetros quedará registrada en el log.

2. Presentar al usuario, a solicitud de este, el contenido del log.

    2.1 Como funcionalidad auxiliar, vinculada a la anterior, modificar el programa visor del fichero de texto que constituye el log.

3. Recibir y registrar en el fichero log los mensajes enviados por el nodo de control con esa finalidad.

Para la primera de las funcionalidades relacionadas, el usuario puede pasar los tres parámetros básicos del sistema bien desde una interface gráfica o bien desde una de texto, ambas muy sencillas, para inmediatamente después retransmitir estos parámetros al nodo de control mediante el método `send(…)` de un objeto instanciado del tipo `MoteIF`.

La revisión del log se encomienda a un programa externo. Por defecto, esta tarea la lleva a cabo el editor *gedit*, aunque puede ser cambiado de acuerdo a la funcionalidad auxiliar vista. La apertura del editor para presentar el contenido del log se consigue mediante la función `exec(…)` de un objeto del tipo `Runtime`, a la que se le pasa el nombre del editor y del fichero a revisar.

Por último, para el registro de los mensajes recibidos en el log, en primer lugar hacemos uso, implementándolo, del método `messageReceived(int to, Message m)`, declarado en la interface `MessageListener` proporcionada por TinyOS en el SDK de Java. Tras la recepción de los mensajes por esta función, se discrimina si corresponden al tipo `SensorsMsg` (que contienen información sobre las lecturas de los sensores y los posibles comandos ordenados) o del tipo `HallMsg`

(información sobre eventos Hall en el nodo actuador), para transcribir al fichero de texto que implementa el log la información correspondiente.

En relación a las clases auxiliares que necesitaremos para manejar los mensajes gestionados por la aplicación, éstas serán construidas por la herramienta MIG de la *toolchain* de TinyOS, que se invoca desde el `Makefile` compartido con la aplicación nesC del nodo de control. A esta herramienta se le pasan como argumentos diferentes parámetros que identifican la localización y el formato de los mensajes a instanciar, así como el lenguaje de programación en el que se implementan estas instancias. Las clases automáticamente generadas por MIG facilitan el acceso a los diferentes componentes de los mensajes gracias a los métodos específicos de acceso a cada campo (recordemos que para ello los mensajes se conforman mediante estructuras de nesC, muy semejantes a las de C).

8.3.1 La interface gráfica:

Una vez construida y probada la aplicación de usuario bajo consola, no resultó complicado el desarrollo de una interface de usuario bajo el editor gráfico del IDE NetBeans 7.1.2. El traslado de la lógica de una a otra aplicación tampoco revistió ningún problema. Dado el limitado tamaño de nuestra aplicación, se ha resuelto su implementación con una sola clase.

9 - MANUAL DE USUARIO: DIRECTRICES

Los códigos fuente de la aplicación deben complementarse con un manual destinado al usuario que le permita dos acciones principales:

1. Compilar con facilidad las aplicaciones involucradas: la aplicación Java para el usuario residente en el PC y las dos aplicaciones TinyOS que residirán en cada una de las *motes*. Además, especificaremos el procedimiento para descargar estas últimas aplicaciones en los nodos.

2. Interactuar y gestionar la aplicación de usuario para lograr los objetivos básicos del proyecto.

En relación a la compilación y descarga de las aplicaciones, la principal estrategia para diseñar los procedimientos ha sido hacerlos lo más sencillos posible para el usuario, aunque no disponga apenas de conocimientos informáticos. Como resumen, se ha actuado en:

- Elaborar un árbol de directorios de las fuentes muy básico, con solo dos directorios: uno para la aplicación del nodo actuador y otro que recoge tanto la aplicación del nodo de control como la aplicación Java para el PC.
- Vinculado con lo anterior, no se utilizan `packages` para no tener que modificar manualmente las clases creadas automáticamente por la herramienta `mig`.
- Resumir los procesos de compilación en `makefile`s adecuados.

No obstante lo anterior, asumiremos que el usuario cuenta con un entorno TinyOS completamente funcional y con permisos suficientes para su explotación, por lo que no se entrará en los procedimientos de instalación del entorno de programación y compilación. Tampoco nos extenderemos en la finalidad y supuestos del sistema de apertura automática de la persiana, que consideramos previamente conocidos por el usuario. En particular, asumiremos la existencia de conexión entre dos salidas del nodo a sendos terminales conectados al motor para transmitirle señales de apertura o cierre de la persiana (y que en el modelo de desarrollo y prototipo simulamos con sendos leds).

Por su parte, se ha intentado que los menús y los mensajes habilitados en la aplicación de usuario sean en buena medida autoexplicativos, por lo que en el manual nos limitaremos a una descripción no excesivamente extensa de las funcionalidades.

En el Anexo 2 transcribimos este manual.

10 - CONCLUSIONES

10.1 Mejoras y posible evolución futura del sistema:

No resultará excesivamente complicada la generalización del prototipo a una red de sensores inalámbricos (WSN) que permita la actuación sobre un buen números de mecanismos que cubran un espacio de superficie (o volumen) importante. Para ello, sería razonable la distribución por zonas de diferentes nodos de control que actuaran, cada uno de ellos, sobre un conjunto de actuadores o persianas. También podría contemplarse la centralización en un único nodo de todos los sensores distribuidos, y que este nodo comandará la actuación de todos los actuadores del sistema. Aunque la diferencia conceptual resulta relevante, la implementación no debería suponer una excesiva carga adicional de trabajo, y la decisión final puede dejarse en función de las características físicas del espacio a gestionar.

Los parámetros implícitos del sistema (factores en las fórmulas de la lógica que decide la actuación sobre el motor) son experimentales y según el entorno en el que se ha desarrollado el prototipo. Además, y por necesidad, la lógica es simple y probablemente no adecuada a todos los contextos. En este sentido queda abierto el desarrollo de una lógica más generalista y con factores más precisos y afinados, tanto teórica como experimentalmente. Esto requeriría, probablemente, de un estudio seguramente multidisciplinar que queda fuera del ámbito de este trabajo.

En cuanto al código, una versión más avanzada de la aplicación gráfica presentaría, como mínimo, dos clases interrelacionadas: una encargada de la presentación de la interface de usuario y otra a cargo de la lógica de la aplicación.

Cabría complementar el sistema con la posibilidad de acceso remoto al mismo vía Internet, para lo que podría resultar útil la utilidad *SerialForwarder*.

10.2 Conclusiones finales:

Nuestro sistema, una vez conectado al mecanismo adecuado, actúa de forma autónoma, y tomando en consideración los factores previstos (temperatura y luminosidad, fundamentalmente), sobre un motor para abrir o cerrar una persiana y permitir así una mayor o menor insolación en un determinado espacio, lo que colaborará a la regulación de su temperatura.

Los materiales hardware y software proporcionados, y una vez estudiados y comprendidos, se han aplicado al desarrollo del prototipo de la manera prevista. Han resultado adecuados y su gestión, desde el punto de vista de los procesos de ingeniería, ha respondido a lo previsible de acuerdo a su documentación básica, hojas de datos, interfaces proporcionadas y documentadas, ejemplos disponibles...

Se ha desarrollado, en opinión del autor, una interface con el usuario sencilla, práctica y auto-explicativa. No obstante, se ha diseñado documentación adicional a la aplicación de usuario que pretende aclarar los puntos posiblemente más problemáticos para un usuario novel: compilación de las fuentes, instalación de las aplicaciones, explicación de los fundamentos de la aplicación de usuario...

En este último sentido de valor añadido para la visión usuario, se ha realizado un esfuerzo de simplicidad en la distribución de los códigos fuente habilitando un árbol de directorios simple, evitando la utilización de `package`s (para que el usuario no tenga que modificar las clases obtenidas automáticamente por MIG), y con unos `makefile`s sencillos pero que cubren la compilación completa de las aplicaciones del sistema (tanto de las basadas en TinyOS como las de Java).

Por último, el desarrollo del proyecto se ha ajustado, con muy ligeras variaciones en el orden de ejecución de algunas tareas, a la planificación inicialmente prevista.

En opinión del autor se han satisfecho con éxito los requisitos y funcionalidades inicialmente propuestos, y se ha hecho dentro del orden que proporciona el seguimiento de los esquemas de los correspondientes procesos de ingeniería.

11 - BIBLIOGRAFIA, DOCUMENTACION

Sobre el hardware del proyecto:

Atmel Corporation. *ATmega1281.*
http://www.atmel.com/devices/atmega1281.aspx

Atmel Corporation (2010). *AT86RF212*.
http://www.atmel.com/Images/doc8168.pdf

Microchip Technology Inc. (2009). *MCP9700/9700A Low-Power Linear Active Thermistor ICs.*
http://ww1.microchip.com/downloads/en/DeviceDoc/21942e.pdf

Sobre TinyOS, nesC, y la instalación del entorno y *toolchain*:

Huber, R.; Fahrni, T. (2009). *Porting the ZigBit 900 Platform to TinyOs*. http://disco.ethz.ch/theses/hs08/mb900_tinyos.pdf

Gay, D. et al. (2003). *nesC 1.1 Language Reference manual.*
http://nescc.sourceforge.net/papers/nesc-ref.pdf

Gay, D. et al. (2003). *The nesC Language: A Holistic Approach to Networked Embedded Systems*. http://www.tinyos.net/papers/nesc.pdf

Levis, P. (2006). *TinyOS Programming.*
http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf

VV.AA. (2011). *Installing TinyOS 2.1.1.*
http://docs.tinyos.net/tinywiki/index.php/Installing_TinyOS_2.1.1#Manual_installation_on_your_host_OS_with_RPMs

Lo, G. (2010). *TinyOS Primer.*
http://www.geoffreylo.com/resources/tinyos-primer/

VV.AA. (2011). *TinyOS Tutorials.*
http://docs.tinyos.net/tinywiki/index.php/TinyOS_Tutorials

Bachmaier, S. A. (2009). *UML 2.0 for modeling TinyOS components.*
http://www.ti5.tu-harburg.de/events/fgsn09/proceedings/fgsn_087.pdf

Sobre el proceso de desarrollo de software:

Jacobson, Booch, Rumbaugh (2000). *El Proceso Unificado de Desarrollo de Software*. Madrid: Addison Wesley Iberoamericana.

Jacobson, Booch, Rumbaugh (2000). *El lenguaje Unificado de Modelado*. Madrid: Addison Wesley Iberoamericana.

Sobre fundamentos de sistemas empotrados y otras fuentes:

VV.AA. (2011). *Sistemas empotrados (material docente UOC).* Barcelona: FUOC.

VV.AA. (2012). *Arp@:Embedded Systems Lab@Home.*
http://cv.uoc.edu/app/mediawiki14/wiki/P%C3%A0gina_principal

VV.AA. *Hall effect sensor; ZigBee; WSN; TinyOS; nesC; Component-oriented programming; Event-driven programming*.
[en|es].wikipedia.org

VV.AA. (2008). *Trabajo final de carrera (material docente UOC).* Barcelona: FUOC.

ANEXOS

- ANEXO 1: UNA NOTA SOBRE EL CALCULO DE LA TEMPERATURA AMBIENTE
- ANEXO 2: MANUAL DEL USUARIO

ANEXO 1: UNA NOTA SOBRE EL CALCULO DE LA TEMPERATURA AMBIENTE

De acuerdo a los esquemas ofrecidos, nuestras *motes* incorporan el sensor de temperatura MCP9700, que ofrece una respuesta prácticamente lineal a las variaciones de la temperatura que registra:

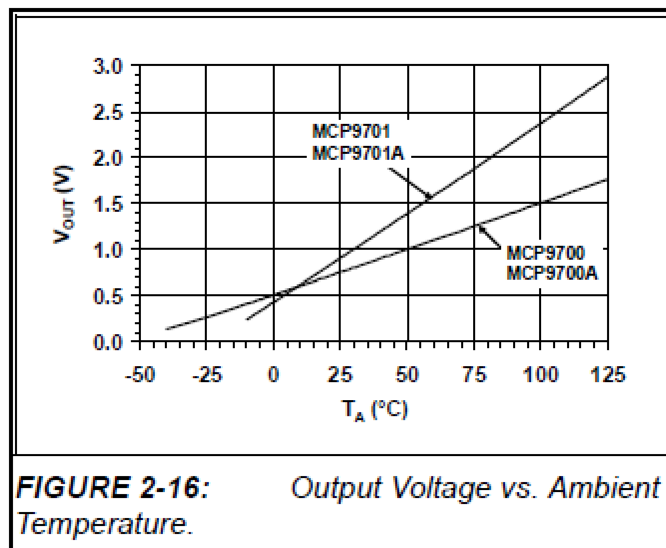


FIGURE 2-16: Output Voltage vs. Ambient Temperature.

Su rango de actuación, según su hoja de datos, está entre los -40º y los 125º centígrados. El coeficiente de temperatura, T_C, registra una variación de 10 mV/Cº, es decir, que una variación de un grado centígrado provoca una variación de 10 mV en el voltaje de salida del sensor. Por último, su función de transferencia es:

$$V_{OUT} = T_C \bullet T_A + V_{0°C}$$

Where:

T_A	=	Ambient Temperature
V_{OUT}	=	Sensor Output Voltage
$V_{0°C}$	=	Sensor Output Voltage at 0°C (See DC Electrical Characteristics table)
T_C	=	Temperature Coefficient (See DC Electrical Characteristics table)

por lo que la temperatura ambiente será $T_A = \frac{V_{OUT} - V_{0°C}}{T_C}$, donde ya sabemos que T_C = 10, y $V_{0°C}$ = 400 mV, según la hoja de datos.

Una vez puesta en funcionamiento la mote obtenemos, para una temperatura ambiente de unos 21º, una lectura del sensor (tras la conversión analógica-digital) de unas 320 unidades, de un rango posible de entre [0, 1023] (ADC de 10 bits).

La curva de respuesta, o la función de transferencia, nos permiten extrapolar que, para la temperatura ambiente comentada y el valor de muestreo obtenido, la salida del sensor rondaría entre 700-720 mV.

Todo lo anterior nos permite apicar una constante de proporcionalidad entre lectura digital y temperatura, para la que obtenemos un valor algo superior a 15 (lectura digital de 320 sobre una temperatura de unos 21º), y que para nuestros fines redondeamos al entero. La temperatura ambiente, por tanto, la obtenemos en nuestra aplicación dividiendo la lectura digital del sensor por 15, lo que nos aportará una precisión sin duda mejorable. pero no obstante útil. Una mayor precisión queda fuera de los objetivos de este trabajo.

ANEXO 2: MANUAL DEL USUARIO

BLIND CONTROL: MANUAL DEL USUARIO

INTRODUCCION:

El presente manual consta de dos partes básicas: los procedimientos de instalación y descarga de las aplicaciones, y la utilización de la aplicación de usuario propiamente dicha para obtener los resultados perseguidos con el sistema. En primer lugar veremos cómo instalar e iniciar la ejecución de las aplicaciones, para posteriormente repasar la interacción con la aplicación de usuario.

Si las aplicaciones ya han sido instaladas y sólo desea conocer cómo funciona la aplicación para regular la apertura o cierre de la persiana automatizada, puede pasar directamente al apartado 2.

1 - COMPILAR E INSTALAR LAS APLICACIONES

1.1 Preliminares:

Usted ha recibido los códigos fuente de las aplicaciones distribuidos en dos subdirectorios (`/ActuadorApp` y `/BlindControl`), ambos situados bajo un directorio raíz `/BlindSrc`. Para la ejecución de la aplicación de usuario (así como para la compilación), será conveniente, aunque no estrictamente necesario, situar el directorio `/BlindControl` al alcance de la variable de entorno `CLASSPATH` (que regula el alcance del entorno de compilación y ejecución de Java), así como del entorno de compilación aportado por TinyOS. Para ello, puede ejecutar el comando:

```
export CLASSPATH=$CLASSPATH:$TOSROOT/apps/BlindSrc/BlindControl
```

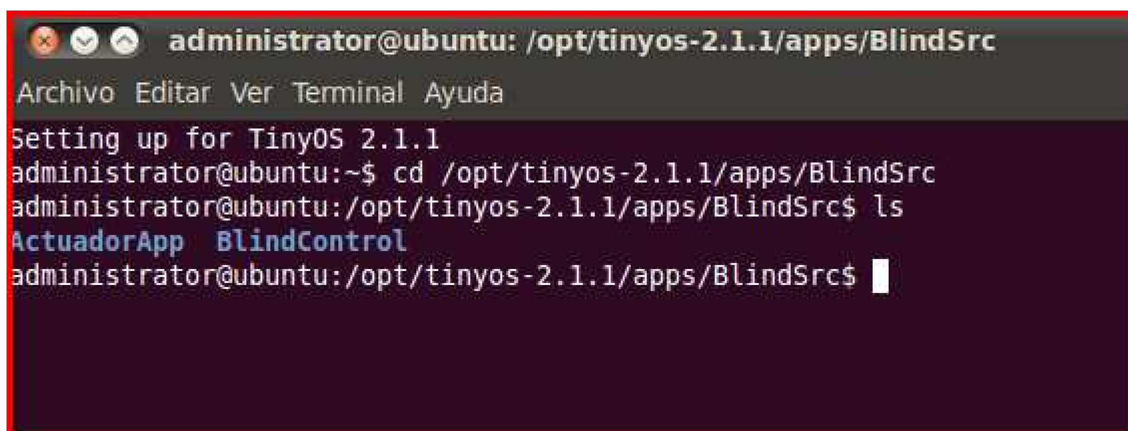
Además, debe contar con permisos para utilizar el puerto serie. Si no fuera así, podría obtenerlos (siempre que tuviera acceso a `root`) mediante el comando:

```
sudo chmod 666 /dev/ttyUSB0
```

En caso de mal funcionamiento o dudas sobre las acciones anteriores, contacte con el administrador de su sistema.

Le recomendamos encarecidamente que sitúe el conjunto de directorios que componen el código fuente de las aplicaciones en el directorio `/apps`, situado en la instalación de TinyOS. Si su versión de este sistema operativo fuera la actual 2.1.1, la ruta al directorio de aplicaciones será `/opt/tinyos-2.1.1/apps`. Todos los ejemplos posteriores asumen la instalación de la aplicación en este directorio.

Una vez copiadas las fuentes al directorio `/apps`, la estructura de directorios resultante debería aparecer como:



1.2 Compilación de la aplicación para el nodo actuador:

Guía rápida:

Conecte la mota que va a implementar el nodo actuador en un conector USB del ordenador donde está compilando la aplicación, y sitúese en el directorio `/ActuadorApp`. Ejecute, por ese orden, los siguientes comandos:

- `make cou24`
- `meshprog -t /dev/ttyUSB0 -f ./build/cou24/main.srec`

Tan pronto se abra el puerto serie (verá una creciente fila de puntos sucesivos en la consola) debe pulsar el botón de *reset* de la placa (el que está junto al conector USB) para descargar la aplicación desde el PC hasta el nodo.

En el último comando se asume que el puerto USB corresponde al dispositivo `ttyUSB0`. Si no fuera el caso, modifique al dispositivo correspondiente. Si obtiene errores, revise el siguiente apartado, *Guía detallada*. Si los problemas persisten o mantiene dudas, contacte con el administrador de su sistema. Tras el último comando, el nodo debería implementar la aplicación y arrancaría su ejecución en unos segundos, siempre y cuando se mantenga la alimentación de la placa.

Guía detallada:

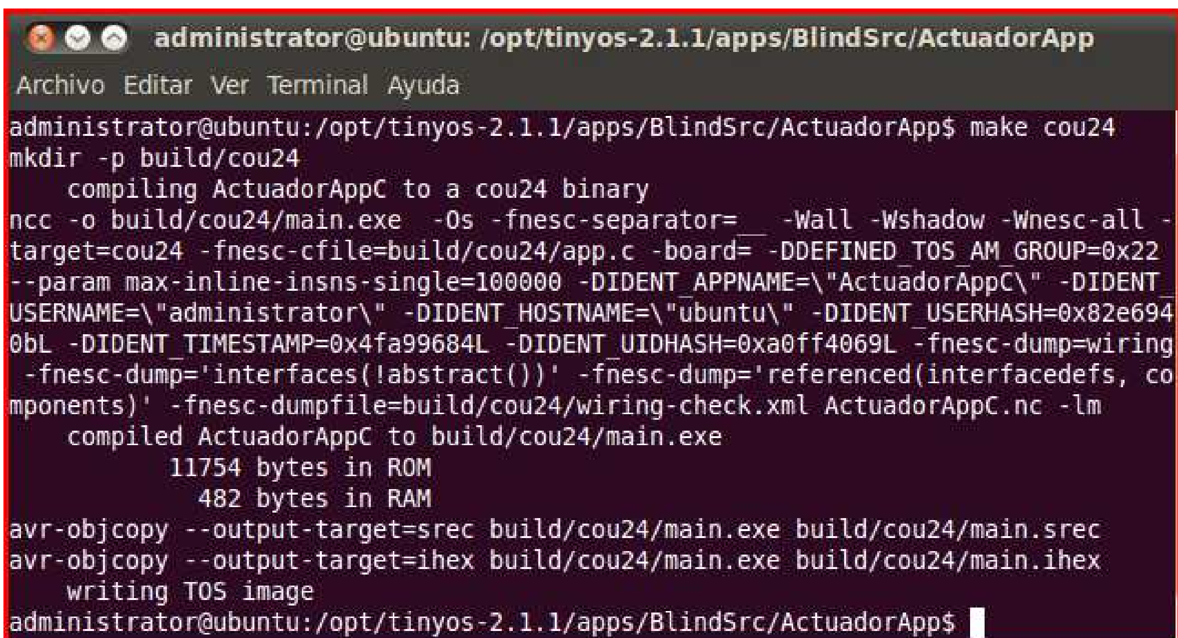
Sitúese en el directorio `/ActuadorApp`, donde debe encontrar los ficheros de la aplicación del nodo actuador (con sufijo `.nc`), un fichero de definiciones (sufijo `.h`), y un `Makefile` que le va a permitir compilar la aplicación del nodo:

```
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc$ cd ActuadorApp
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/ActuadorApp$ ls
ActuadorAppC.nc  GestionMotorP.nc       Makefile
ActuadorApp.h    GestionSensorHallP.nc
```

Para compilar, desde este directorio debe ejecutar el comado `make` pasándole como parámetro la plataforma hardware cou24:

```
make cou24
```

Obtendrá:



Lo anterior habrá generado ficheros compilados que habrán quedado situados en los directorios `/build/cou24`, creados al efecto.

Una vez compilada la aplicación, debe descargarse en la mota que ejecutará las funciones de nodo actuador. Para ello, utilizaremos la utilidad `meshprog` del entorno TinyOS, pasándole como parámetros tanto el dispositivo USB al que estará conectada la placa como el fichero con la imagen de la aplicación a descargar.

Por lo tanto, y si no se ha hecho ya, conecte la mota a un puerto USB del ordenador en el que esté ejecutando la compilación. Aunque la conexión

física es mediante un puerto USB, la conexión lógica se implementa como un puerto serie, por lo que en condiciones usuales, y si no tiene habilitado otro puerto serie, el dispositivo activado por defecto corresponderá al `/dev/ttyUSB0`. Si no fuera así, deberá cerciorarse de qué puerto es el utilizado por la conexión de la placa. En caso de problemas o dudas, contacte con el administrador de su sistema. En definitiva, el comando `meshprog` para volcar la aplicación a la placa será:

```
meshprog -t /dev/ttyUSB0 -f ./build/cou24/main.srec
```

Tan pronto quede abierto el puerto serie (mensaje `Opened device /dev/ttyUSB0`) y se represente en la consola una secuencia creciente de puntos sucesivos, debe pulsar el botón de *reset* de la placa (que está junto al conector USB) para descargar la aplicación. Con este proceso obtendríamos la siguiente salida en la consola:

```
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/ActuadorApp$ meshprog -t /d
ev/ttyUSB0 -f ./build/cou24/main.srec
using tty /dev/ttyUSB0
reading from file ./build/cou24/main.srec
sending ./build/cou24/main.srec -> /dev/ttyUSB0

Opened file ./build/cou24/main.srec
open: Device or resource busy
Failed on device /dev/ttyUSB0
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/ActuadorApp$ meshprog -t /d
ev/ttyUSB0 -f ./build/cou24/main.srec
using tty /dev/ttyUSB0
reading from file ./build/cou24/main.srec
sending ./build/cou24/main.srec -> /dev/ttyUSB0

Opened file ./build/cou24/main.srec
Opened device /dev/ttyUSB0
...........................,[i��&]
Starting transmission.
finished!
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/ActuadorApp$
```

Una vez descargada la aplicación, y si se mantiene la alimentación de la placa (bien por baterías, bien por la propia alimentación del puerto USB), comenzará a ejecutarse a los pocos segundos, lo que se podrá comprobar por el parpadeo del led verde.

1.3 Compilación de la aplicación para el nodo de control y la aplicación de usuario:

Guía rápida:

Conecte la mota que va a implementar el nodo de control en un conector USB del ordenador donde está compilando la aplicación, y sitúese en el

directorio `/BlindControl`. Ejecute, por ese orden, los siguientes comandos:

- `make cou24`
- `meshprog -t /dev/ttyUSB0 -f ./build/cou24/main.srec`

Tan pronto se abra el puerto serie (verá una creciente fila de puntos sucesivos en la consola) debe pulsar el botón de *reset* de la placa (el que está junto al conector USB) para descargar la aplicación desde el PC hasta el nodo.

En el último comando se asume que el puerto USB corresponde al dispositivo `ttyUSB0`. Si no fuera el caso, modifique al dispositivo correspondiente. Si obtiene errores al compilar, especialmente con Java, revise el apartado *1.1 Preliminares* de este capítulo, y/o el siguiente apartado, *Guía detallada*. Si los problemas persisten o mantiene dudas, contacte con el administrador de su sistema. Tras el último comando, el nodo debería implementar la aplicación y arrancaría su ejecución en unos segundos, siempre y cuando se mantenga la alimentación de la placa.

Guía detallada:

Para mayor simplicidad y rapidez en el procedimiento de compilación, se distribuyen en el mismo directorio los ficheros fuente de la aplicación del nodo de control y de la aplicación de usuario.

Sitúese en el directorio `BlindControl`, donde debe encontrar los ficheros de la aplicación del nodo de control (con sufijo `.nc`), un fichero de definiciones (sufijo `.h`), un fichero (o dos si se incluye en la distribución la aplicación bajo interface gráfica) correspondiente a la aplicación de usuario (sufijo `.java`), y un `Makefile` que le va a permitir compilar tanto la aplicación del nodo como la aplicación de usuario:

```
administrator@ubuntu:/opt/tinyos-2.1.1/apps$ cd BlindSrc
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc$ ls
ActuadorApp  BlindControl
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc$ cd BlindControl
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/BlindControl$ ls
ActualizarLogP.nc   EnviarComandoActuadorP.nc   IFSendCommand.nc
BlindControl.java   GestionParametrizacionP.nc  IFUpdateLog.nc
ControlAppC.nc      GestionSensoresEntornoP.nc  IFUpdateParamControl.nc
ControlApp.h        GestionSensorHallP.nc       Makefile
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/BlindControl$
```

Para compilar, desde este directorio debe ejecutar el comado `make` pasándole como parámetro la plataforma hardware cou24:

```
make cou24
```

Obtendrá, si todo transcurre correctamente, algo muy semejante a lo siguiente:

```
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/BlindControl$ make cou24
mkdir -p build/cou24
mig java -target=null -I/opt/tinyos-2.1.1/tos/lib/T2Hack -DIDENT_APPNAME=\"Contr
olAppC\" -DIDENT_USERNAME=\"administrator\" -DIDENT_HOSTNAME=\"ubuntu\" -DIDENT_
USERHASH=0x82e6940bL -DIDENT_TIMESTAMP=0x4faad72cL -DIDENT_UIDHASH=0x845505acL -
java-classname=HallMsg ControlApp.h HallMsg -o HallMsg.java
mig java -target=null -I/opt/tinyos-2.1.1/tos/lib/T2Hack -DIDENT_APPNAME=\"Contr
olAppC\" -DIDENT_USERNAME=\"administrator\" -DIDENT_HOSTNAME=\"ubuntu\" -DIDENT_
USERHASH=0x82e6940bL -DIDENT_TIMESTAMP=0x4faad72cL -DIDENT_UIDHASH=0x845505acL -
java-classname=ParamMsg ControlApp.h ParamMsg -o ParamMsg.java
mig java -target=null -I/opt/tinyos-2.1.1/tos/lib/T2Hack -DIDENT_APPNAME=\"Contr
olAppC\" -DIDENT_USERNAME=\"administrator\" -DIDENT_HOSTNAME=\"ubuntu\" -DIDENT_
USERHASH=0x82e6940bL -DIDENT_TIMESTAMP=0x4faad72cL -DIDENT_UIDHASH=0x845505acL -
java-classname=SensorsMsg ControlApp.h SensorsMsg -o SensorsMsg.java
javac -target 1.6 -source 1.6 *.java
    compiling ControlAppC to a cou24 binary
ncc -o build/cou24/main.exe  -Os -fnesc-separator=__ -Wall -Wshadow -Wnesc-all -
target=cou24 -fnesc-cfile=build/cou24/app.c -board= -DDEFINED_TOS_AM_GROUP=0x22
--param max-inline-insns-single=100000 -I/opt/tinyos-2.1.1/tos/lib/T2Hack -DIDEN
T_APPNAME=\"ControlAppC\" -DIDENT_USERNAME=\"administrator\" -DIDENT_HOSTNAME=\"
ubuntu\" -DIDENT_USERHASH=0x82e6940bL -DIDENT_TIMESTAMP=0x4faad72cL -DIDENT_UIDH
ASH=0x845505acL -fnesc-dump=wiring -fnesc-dump='interfaces(!abstract())' -fnesc-
dump='referenced(interfacedefs, components)' -fnesc-dumpfile=build/cou24/wiring-
check.xml ControlAppC.nc -lm
    compiled ControlAppC to build/cou24/main.exe
           18104 bytes in ROM
             834 bytes in RAM
avr-objcopy --output-target=srec build/cou24/main.exe build/cou24/main.srec
avr-objcopy --output-target=ihex build/cou24/main.exe build/cou24/main.ihex
    writing TOS image
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/BlindControl$
```

El proceso anterior habrá generado ficheros compilados de la aplicación del nodo de control y clases de la aplicación de usuario, que habrán quedado distribuidos tanto en el directorio `/build/cou24`, creado al efecto para recoger la aplicación del nodo, como en el propio directorio `BlindControl` para las clases de la aplicación java del usuario.

Puede obtener errores del código Java al compilarlo si su sistema no permite la visibilidad de Java sobre el propio directorio de trabajo, aunque esta posibilidad es cada vez menos frecuente en los actuales entornos del lenguaje. De darse este caso, diríjase al apartado *1.1 Preliminares* de este capítulo, donde podrá revisar cómo hacer visible un directorio a Java por medio de la variable de entorno `CLASSPATH`.

Una vez compiladas las aplicaciones, la que se va a implementar en el nodo de control debe descargarse en la mota que ejecutará tales funciones. Para ello, utilizaremos la utilidad `meshprog` del entorno

TinyOS, pasándole como parámetros tanto el dispositivo USB al que estará conectada la placa como el fichero con la imagen de la aplicación a descargar.

Por lo tanto, y si no se ha hecho ya, conecte la mota a un puerto USB del ordenador en el que esté ejecutando la compilación. Aunque la conexión física es mediante un puerto USB, la conexión lógica se implementa como un puerto serie, por lo que en condiciones usuales, y si no tiene habilitado otro puerto serie, el dispositivo activado por defecto corresponderá al `/dev/ttyUSB0`. Si no fuera así, deberá cerciorarse de qué puerto es el utilizado por la conexión de la placa. En caso de problemas o dudas, contacte con el administrador de su sistema. En definitiva, el comando `meshprog` para volcar la aplicación a la placa será:

```
meshprog -t /dev/ttyUSB0 -f ./build/cou24/main.srec
```

Tan pronto quede abierto el puerto serie (mensaje `Opened device /dev/ttyUSB0`) y se represente en la consola una secuencia creciente de puntos sucesivos, debe pulsar el botón de *reset* de la placa (que está junto al conector USB) para descargar la aplicación. Con este proceso obtendríamos la siguiente salida en la consola:

```
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/BlindControl$ meshprog -t /
dev/ttyUSB0 -f ./build/cou24/main.srec
using tty /dev/ttyUSB0
reading from file ./build/cou24/main.srec
sending ./build/cou24/main.srec -> /dev/ttyUSB0

Opened file ./build/cou24/main.srec
Opened device /dev/ttyUSB0
,[~E]
.........................,[i��&]
Starting transmission.
finished!
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/BlindControl$
```

Una vez descargada la aplicación, y si se mantiene la alimentación de la placa (bien por baterías, bien por la propia alimentación del puerto USB), comenzará a ejecutarse a los pocos segundos, lo que se podrá comprobar por el parpadeo del led verde.

En cuanto a la aplicación de usuario, la clase principal es `BlindControl.class`, que hará uso de otras clases auxiliares, pero necesarias, para desarrollar sus funciones. La ejecución de la aplicación se obtiene mediante el comando:

```
java BlindControl
```

Ver apartados posteriores sobre la gestión e interacción con la aplicación (capítulo 2), así como su posible ejecución bajo una interface gráfica más cómoda (apartado 2.6).

Recuerde que aunque ahora es usual que Java pueda ejecutar clases desde el mismo directorio en el que radican, en algunos sistemas no es posible, por lo que es recomendable que el directorio donde finalmente se emplacen las clases esté bajo el alcance de la variable de entorno `CLASSPATH`. Como ejemplo, si no pudiéramos arrancar la aplicación de usuario con el comando visto en el párrafo anterior, podemos redefinir `CLASSPATH` para que incluya el directorio de instalación por defecto mediante el comando:

```
export CLASSPATH=$CLASSPATH:$TOSROOT/apps/BlindSrc/BlindControl
```

tal y como ya hemos visto en el apartado *1.1 Preliminares*.

Si tiene conocimientos de cómo el entorno Java accede y permite la ejecución de clases, puede llevar los ficheros de la aplicación de usuario (todos los ficheros con sufijo `.class`) a otro directorio de su elección, contemplando, en su caso, los matices del párrafo anterior. Si tiene problemas o dudas, contacte con el administrador de su sistema.

1.4 Instalación del hardware

El nodo actuador deberá ser colocado en el espacio habilitado para el mismo en el actuador de la persiana móvil. Su alimentación debe realizarse mediante dos pilas de 1,5 V tipo AA. Para cualquier aclaración o duda, contacte con el instalador del sistema motorizado.

El nodo de control deberá permanecer conectado al puerto USB correspondiente del PC que implementa la aplicación de usuario. Su alimentación puede dejarse a cargo de la propia conexión USB. Para la correcta interactuación entre este nodo y la aplicación de usuario, ésta debe estar arrancada, por lo que se le habrán pasado, al inicio, los parámetros básicos del sistema.

1.5 Posibles tareas de limpieza tras la compilación e instalación:

Aunque es recomendable el mantenimiento de los códigos fuente en los directorios instalados, lo que permitirá una recompilación rápida llegado el caso, una vez compiladas las aplicaciones, y de acuerdo a los intereses del usuario, pueden efectuarse las siguientes acciones para reducir el número de ficheros o directorios ociosos:

- Aplicación del nodo de actuador: todo el código bajo el directorio /ActuadorApp puede eliminarse. También puede eliminarse el propio directorio.
- Aplicación del nodo de control: todo el código bajo el directorio /BlindControl con sufijos .nc y .h puede eliminarse. También el fichero Makefile.
- Aplicación de usuario: todo el código bajo el directorio /BlindControl con sufijo .java puede eliminarse.

Si llega a realizarse todo lo anterior, del contenido de los directorios tras la compilación sólo quedarían las clases de la aplicación de usuario bajo Java (ficheros con sufijos .class) dentro del directorio BlindControl. Si éstas, de acuerdo a lo visto en el apartado anterior, se trasladaran a otro directorio, todo el árbol de directorios colgando del directorio /BlindSrc, incluyendo esta raíz, podría eliminarse.

En el directorio /BlindControl puede realizarse una limpieza más específica ejecutando el comando make clean, que eliminará todos los directorios y ficheros creados al compilar, dejando únicamente los ficheros fuente iniciales. Tenga en cuenta que con esta acción también se eliminarían las clases Java, lo que equivale a eliminar la aplicación de usuario.

2 - MANUAL DE USUARIO APLICACIÓN PC: BlindControl

2.1 Introducción:

La aplicación de usuario *BlindControl* le va a permitir gestionar en lo necesario el sistema de apertura automática e inteligente de sus persianas motorizadas, que actuará en función de la temperatura ambiente y la luminosidad detectadas por los sensores habilitados al efecto. Para ello, podrá actuar sobre dos acciones principales y otra auxiliar:

- La fijación de los parámetros básicos del sistemas: el rango de temperaturas normales o habituales dentro del que se pretende mantener la temperatura ambiente, y la frecuencia de muestreo de los sensores y, por tanto, de actuación de la lógica de la aplicación para actuar (o no) sobre el motor de la persiana.
- La revisión del log del sistema, en el que quedan reflejados los muestreos periódicos, los comandos (apertura o cierre de la persiana) decididos por la aplicación, o la indicación de si la persiana está totalmente cerrada o no.
- Como funcionalidad auxiliar, se puede pasar a la aplicación el editor bajo el que se quiere revisar el log.

2.2 Arranque y primeras acciones de la aplicación de usuario:

Una vez instalada la aplicación de usuario (para detalles, ver apartado *1.3 Compilación de la aplicación para el nodo de control y la aplicación de usuario* del capítulo anterior), su arranque lo efectuamos desde el directorio donde se encuentran las clases (o desde cualquier otro si el directorio de instalación está bajo el alcance de la variable de entorno CLASSPATH) mediante el comando `java BlindControl`, con el que accederemos al menú de introducción inicial de parámetros del sistema (tome nota de que en el apartado 2.6 podrá revisar una forma alternativa de ejecución de la aplicación bajo una interface gráfica):

```
administrator@ubuntu:/opt/tinyos-2.1.1/apps/BlindSrc/BlindControl$ java BlindCon
trol
serial@/dev/ttyUSB0:57600: resynchronising

-------------------------------------------
C O N T R O L   P E R S I A N A   M O V I L
-------------------------------------------
     == INTRODUCCION DE PARAMETROS ==

> Introduzca temperatura mínima:
```

Dados los requisitos a satisfacer, la estrategia de la aplicación es ejecutarse de forma indefinida mediante un bucle de control muy sencillo, por lo que en las raras ocasiones en las que se reinicie habrá que introducir, como primera medida, los parámetros del rango de temperaturas bajo el que actuará la lógica de la aplicación para determinar acciones sobre la persiana, así como la frecuencia (en minutos) con la que se muestrearán los sensores de temperatura y luminosidad.

Una vez introducidos estos parámetros, obtendremos mensajes del sistema con los resultados que se van obteniendo, ofreciéndonos la continuación de la interface (hacia el menú principal) pulsando la tecla *intro*:

```
-------------------------------------------
C O N T R O L   P E R S I A N A   M O V I L
-------------------------------------------
     == INTRODUCCION DE PARAMETROS ==

> Introduzca temperatura mínima:
21
> Introduzca temperatura máxima:
25
> Introduzca frecuencia muestreo:
15
  --Introducción de parámetros finalizada
  --Enviando mensaje a nodo control con los nuevos parámetros...
  --Nuevos parámetros del sistema fijados
> Pulse intro para continuar
```

Tras continuar, lo anterior nos llevará al menú principal de la aplicación:

```
-------------------------------------------
C O N T R O L   P E R S I A N A   M O V I L
-------------------------------------------

           == MENU PRINCIPAL ==

PARAMETROS ACTUALES:

    Temperatura mínima:  21
    Temperatura máxima:  25
    Frecuencia muestreo: 15

    Visor del log: gedit

OPCIONES DEL PROGRAMA:
    1 - Revisar log
    2 - Modificar parámetros
    3 - Cambiar editor para revisar log


> Introduzca opción [1, 2 o 3]:
```

Como vemos, en la consola se identifican de forma inmediata dos partes claramente diferenciadas: en una primera se recuerdan los parámetros vigentes del sistema, mientras que en la segunda se identifican de forma directa las tres opciones que llevan a las tres funcionalidades del sistema que hemos visto en el apartado *2.1 Introducción*: la revisión del log donde la aplicación registra los principales eventos, la modificación de los parámetros básicos del sistema, y la elección del editor para la revisión del log.

2.3 Revisión del log de la aplicación:

La elección de esta funcionalidad introduciendo la opción '1' en la aplicación de consola, nos abrirá en el editor correspondiente el log del sistema (ver próximo apartado 2.5, donde se amplía información sobre este editor).

Una salida de ejemplo en el editor *gedit* podría resultar la siguiente (note que en este ejemplo la frecuencia de muestreo es de 10 segundos, lo que no tiene sentido a efectos prácticos puesto que las posibles variaciones de temperatura siempre resultarán en periodos mucho más dilatados; la frecuencia de muestreo por defecto del sistema es de 15 minutos):

```
Sat May 12 08:18:14 PDT 2012 - COUNTER: 12  C_PHOTO: 242  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
Sat May 12 08:18:24 PDT 2012 - COUNTER: 13  C_PHOTO: 245  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
Sat May 12 08:18:24 PDT 2012 - Nuevos parámetros:  tempMax = 24; tempMin = 21; freq = 15
Sat May 12 08:18:33 PDT 2012 - COUNTER: 14  C_PHOTO: 243  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
Sat May 12 08:18:34 PDT 2012 - HALL EVENT número: 5:  Persiana cerrada
Sat May 12 08:18:43 PDT 2012 - COUNTER: 15  C_PHOTO: 248  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
Sat May 12 08:18:53 PDT 2012 - COUNTER: 16  C_PHOTO: 246  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
Sat May 12 08:19:03 PDT 2012 - COUNTER: 17  C_PHOTO: 238  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
Sat May 12 08:19:12 PDT 2012 - COUNTER: 18  C_PHOTO: 248  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
Sat May 12 08:19:21 PDT 2012 - HALL EVENT número: 6 Persiana no cerrada
Sat May 12 08:19:22 PDT 2012 - COUNTER: 19  C_PHOTO: 247  TEMP: 20ºC  moteId: 1
                               COMMAND: No hay comando
```

En el log del sistema se registran los siguientes eventos:

- Las medidas obtenidas del muestreo de los sensores de temperatura e iluminación. Estas medidas se realizan con la frecuencia determinada por el usuario al arrancar la aplicación, o bien por las modificaciones que pueden realizarse utilizando la opción *2 – Modificar parámetros*, que veremos en el siguiente apartado.
- Tras cada muestreo, el sistema decide si actúa sobre el motor de la persiana, o no, para intentar devolver la temperatura al rango estipulado (también vía parámetros elegidos por el usuario). Estos comandos de actuación sobre la persiana (abrirla o cerrarla), así como el tiempo durante el cual el motor tiene que actuar (o la no necesidad de actuación), son registrados en el log.
- Una indicación del grado de cierre de la persiana. Las características de registro de esta opción (basada en un sensor de efecto Hall que queda enfrentado a un imán cuando la persiana está completamente cerrada, por lo que el sensor devuelve un pico de voltaje cuando imán y sensor quedan enfrentados y, de nuevo, cuando se separan) sólo permiten discriminar cuándo está completamente cerrada la persiana y cuándo no. Por tanto, quedarán registrados en el log los momentos en que la persiana queda completamente cerrada y cuando se empieza a abrir.
- Los cambios en los parámetros del sistema que incluye el usuario quedan también registrados en el log.

Para volver al menú inicial deberemos pulsar intro, según se nos recordará en el correspondiente mensaje de la consola.

2.4 Modificación de los parámetros del sistema:

Como hemos visto, nada más arrancar la aplicación de usuario éste tendrá que introducir los parámetros del sistema. La opción que nos ocupa ahora nos permitirá modificarlos cuando sea conveniente y cuantas veces se considere necesario.

La pantalla y funcionalidad de esta opción son las mismas que hemos visto en el apartado *2.2 Arranque y primeras acciones de la aplicación de usuario* cuando hemos tratado sobre la introducción inicial de los parámetros, por lo que para este punto nos remitimos a dicho apartado.

Tras la introducción de los parámetros, y para volver al menú inicial, deberemos pulsar intro, según se nos recordará en el correspondiente mensaje de la consola.

2.5 Modificar el visor/editor del log:

Por defecto, la aplicación se servirá del editor *gedit* para presentar el fichero de texto soporte del log del sistema. Este editor puede cambiarse eligiendo la opción '3' de las que se presentan en el menú principal.

La pantalla que se nos presenta nos solicita la introducción del nuevo programa, que una vez registrado será utilizado en adelante para esta función por la aplicación:

```
-------------------------------------------
C O N T R O L   P E R S I A N A   M O V I L
-------------------------------------------
       == INTRODUCCION DE EDITOR ==

> Introduzca nombre editor (nombre del ejecutable):
vim
  --Nuevo editor del sistema fijado:  vim
> Pulse intro para continuar
```

Podrá utilizarse cualquier editor siempre que su ejecutable permita el arranque (en una nueva ventana) desde consola y pasándole como parámetro el fichero a abrir.

Para volver al menú inicial deberemos pulsar intro, según se nos recuerda en el correspondiente mensaje de la consola.

2.6 La aplicación gráfica:

La versión definitiva de esta aplicación incluye la opción de ejecutarla bajo una interface gráfica, más sencilla en su uso que la que acabamos de ver en los apartados anteriores y más agradable visualmente.

Para arrancar esta versión, y asumiendo los mismos requisitos previos que ya hemos revisado (además de considerar la existencia de un escritorio gráfico en su sistema operativo), debe introducir el comando `java BlindControlGUI`, que ejecutará la aplicación bajo la siguiente ventana de gestión:

Su uso es obvio y prácticamente inmediato, y nos remitimos a los apartados anteriores para detalles sobre cada una de las funcionalidades.