Plataforma WSN/NIMS III

Programación básica de motas

J. Jiménez
A. Priegue
J. Piazzese
E. Oñate

26 Octubre 2005

ÍNDICE

Introducción a NesC

NesC es un lenguaje de programación que se utiliza para crear aplicaciones que serán ejecutadas en sensores que ejecuten el sistema operativo de *TinyOs*, por tanto dicho lenguaje proporciona ciertas características necesarias para poder realizar aplicaciones de una forma más cómoda para el programador.

Concretamente, se basa en una programación orientada a componentes, esto es, una aplicación se crea ensamblando componentes, esta filosofía permite abstraer al programador de bastantes detalles de baja implementación presentes en el sistema operativo.

La idea que hay detrás de este tipo de programación, es que el propio sistema operativo en conjunción con las casas que venden los dispositivos de sensores proporcionan de forma intrínseca ciertos componentes ya implementados que ofrecen al programador una serie de funciones y utilidades para que pueda utilizar dichos componentes y centrarse sólo en programar la funcionalidad que desea en el dispositivo sin necesidad de tener que preocuparse por todos estos aspectos.

Por otro lado, *NesC* combina ciertos aspectos de la orientación a objetos, en el sentido de que se basa en una programación orientada a interfaces y de la orientación a eventos, en el sentido de que posee un manejo de eventos propio de este tipo de lenguajes como Visual Basic.

Todos los componentes que ofrece de forma intrínseca el sistema operativo se van a denominar a partir de ahora **componentes primitivos** y los componentes proporcionados por terceros, contribuciones, librerías, aplicaciones se van a denominar **componentes complejos**.

Estos componentes, ya sean primitivos o complejos proporcionan unas interfaces, y si un programador quiere utilizar un componente, la forma de hacerlo es usar dichas interfaces, pero recordemos que son interfaces en el sentido OO, por lo que en un momento dado se podría cambiar un componente por otro siempre y cuando este otro proporcionase la misma interfaz y este cambio no afectaría al código de la implementación de la aplicación.

De esta explicación se puede entresacar, que se van a tener dos partes diferentes como mínimo para un componente, la parte de implementación que estará programada hacia componentes y la parte de configuración que permitirá decidir que componentes son los que estoy utilizando para proporcionar dichas interfaces a mi componente, esto como veremos más adelante se denomina **wiring.**

En cuanto a la orientación a eventos, básicamente la forma de programar la aplicación no es del todo secuencial, sino que atiende a una programación basada en eventos, de manera que se programan las acciones que se desea realizar cuando se produzca un evento y este fragmento se ejecutará exactamente cada vez que se lleve a cabo dicho evento.

Estructura de un Componente

Un componente desde el punto de vista de programación está compuesto por varias secciones y el conjunto de todas ellas dan lugar a la creación de dicho componente.

Por tanto, primero, vamos a empezar definiendo el convenio que es utilizado para organizar dichas secciones por *TinyOS*. En general, un componente posee tres grandes secciones que son: ***Configuration, Implementation, Module***. Estas tres secciones han de estar obligatoriamente presentes en cualquier componente aunque puedan estar vacías.

El estándar de *TinyOS* determina, que las secciones de *Configuration e Implementation* han de ir en un fichero que recibirá el nombre del componente con la extensión ***.nc*** y la tercera sección de *Module* deberá ir en otro fichero aparte que recibirá el nombre del componente concatenado con una **M** mayúscula (la M da el significado al fichero, es el significado de *Module*), este último fichero también poseerá a extensión *.nc*.

Otra buena costumbre consiste en crear un fichero de *header* o cabecera con extensión *.h* que contenga todas las enumeraciones, registros o tipos de datos creados por el usuario de los que hace uso la aplicación, y cuando se realiza esto la forma de ligar dicho fichero con los otros dos es utilizando al principio de los otros ficheros la directiva *includes header;* aunque como mención especial decir que si nos fijamos mejor en esta directiva se puede ver que no se incorpora la extensión *.h* en la misma.

A continuación se verá con más detalle cada una de las secciones mencionadas anteriormente.

Implementation

Esta sección se va a encargar de definir las conexiones que hay entre los diferentes componentes que utiliza la aplicación, esto es debido a que si recordamos un poco, se ha comentado que la programación de un componente (que se llevará a cabo en la sección de *module*) se hace utilizando interfaces y dichas interfaces para poder utilizarlas las ha de proporcionar un componente, entonces básicamente en esta sección se definen cuales son los componentes que proporcionan las interfaces a nuestra aplicación (por lo general serán *componentes primitivos*).

Una vez que conocemos la finalidad de esta sección y llegados a este punto, vamos a insertar un concepto nuevo que es la diferencia que existe entre una aplicación que está ya disponible para ser ejecutada en un sensor y un componente cualquiera. La diferencia es muy poca, y consiste en que una aplicación es un componente como cualquier cosa en este lenguaje que en su sección de implementación hace uso de un componente especial denominado *Main*.

Bueno, profundizando un poco más si necesitamos crear un componente para un sensor, por lo general tenemos que utilizar las interfaces que nos proporcionan otros *componentes primitivos o no* y en definitiva para cada una de estas interfaces que nosotros utilicemos en la creación de nuestro componente se han de definir obligatoriamente relaciones con las componentes que proporcionen dichas interfaces, al proceso de definir estas relaciones se le conoce como **wiring**.

Ahora veremos cómo podemos definir estas relaciones, la sección de implementación consta de dos partes, la primera es una declaración de intenciones en la que se especifican todos los componentes que se van a utilizar en nuestra aplicación. Esta declaración se realiza mediante la palabra reservada *components* después de la cual van separados por una coma simple todos los componentes.

La segunda parte de esta sección corresponde a la definición de las relaciones que hay entre las interfaces que utiliza nuestra aplicación y las interfaces que proporcionan los componentes; y la forma de definir que la interfaz que proporciona un componente es la que se corresponde a la que estoy utilizando en mi aplicación es la siguiente:

componente.interfaz -> miaplicación.interfaz

Pero ahora nos surge un problema y se da en el caso en que mi aplicación desee utilizar dos interfaces iguales, es decir que poseen el mismo nombre, pero que están proporcionadas por componentes diferentes, pues bien, para que el lenguaje pueda permitir esto se puede utilizar alias para las interfaces (dichos alias se especificarán en la sección *module*) y la forma de indicar en esta sección que la interfaz que proporciona un componente corresponde a una de las interfaces que utiliza mi aplicación sería:

componente.interfaz -> miaplicación.Aliasinterfaz

Ahora bien, si queremos realizar una aplicación ejecutable en un sensor, como se ha indicado anteriormente se ha de utilizar el componente especial *Main* dicho componente proporciona una interfaz especial denominada *StdControl* y por tanto mi aplicación

obligatoriamente ha de proporcionar dicha interfaz, aunque la manera de proporcionarla la veremos cuando lleguemos a la correspondiente sección, vamos a ver ahora como tendríamos que hacer el *wiring* y así nos valdrá como primer ejemplo.

```
implementation {
        components Main, MiAplicacionM;
        Main.StdControl -> MiAplicacionM.StdControl;
}
```

Ahora veámoslo desde otra perspectiva, imaginemos que la única información que poseo es la que aparece en el ejemplo de arriba, de éste yo podría asegurar que mi aplicación proporciona la interfaz *StdControl* y que además no utiliza ninguna otra por que si no debería de haber realizado el *wiring* del componente que proporcione esa interfaz con la interfaz de mi aplicación, lo que implica también que dicho componente debería de estar declarado en la directiva *components*.

Otra cosa que se puede dar en esta sección es que si poseemos un componente A que utiliza la interfaz IC que proporciona un componente B y el componente B utiliza la interfaz IC que proporciona el componente A, esto lo podemos resolver de dos formas la primera es con lo que ya sabemos lo que quedaría:

```
implementation {
        components A, B;
        A.IC -> B.IC;
        B-IC -> A.IC;
 }
```

Sin embargo por motivos de claridad y para que la sección de *wiring* se haga lo más legible y entendible posible, el lenguaje *NesC* ofrece otro operador de conexión par dar esta semántica, es el operador =. Con lo que quedaría de la siguiente forma:

```
implementation {
        components A, B;
        A.IC = B.IC;
}
```

Introduzcamos ahora otro concepto muy utilizado en el *wiring* y que es vital para la correcta comprensión del mismo, además proporciona bastante potencia al lenguaje, es el concepto de *interfaces paramétricas*, para entender su significado se va a empezar por poner un caso real, pensemos en un componente que se utiliza para recibir paquetes por el medio físico del aire, dicho componente proporciona una interfaz denominada *ReceiveMsg* para la recepción, y dicha interfaz obliga a quien la utilice a programa un evento que se producirá cuando se reciba un paquete.

Ahora bien, pensemos que tenemos dos tipos de paquetes, uno que es de nuestra aplicación y otro que es de una aplicación que hemos comprado a terceros y estas dos aplicaciones han de coexistir en el mismo medio físico (el aire) entonces cómo informo yo al componente que cuando quiero recibir un paquete, me refiero a los paquete que son de mi aplicación, pues bien. para ello se utilizan las *interfaces paramétricas*, yo me crearé un identificador para mi tipo de mensaje que sea diferente al identificador utilizado para el otro tipo e informaré en la sección de *wiring* que utilizo la interfaz pero para este tipo de paquetes.

El concepto que va por debajo de esto, es que el lenguaje de programación mediante la utilización de *interfaces paramétricas*, permite tener varias instancias de una interfaz proporcionadas por un mismo componente.

La forma de indicarlo sería la siguiente:

```
implementation {
        components GenericComm, //Permite envio/recepcion de paquetes
        MiAplicacion;
        MiAplicacion.RecibeMsg -> GenericComm.RecibeMsg[MI_MENSAJE];
}
```

Configuration

Esta sección del componente ha de estar obligatoriamente presente pero sólo contendrá algo en el caso en el que se pretenda crear un componente, no mediante su implementación de código directa (en la sección *Module),* sino mediante la composición de otros componentes ya creados, es un mecanismo que ofrece el lenguaje de programación para poder crear un nuevo componente mediante la combinación directa de otros.

La estructura de esta sección es la misma que la de la sección de *Implementación* que acabamos de ver, además posee la misma semántica, un ejemplo podría ser:

```
Configuration MiAplicacion {
        implementation {
                components GenericComm, //Permite envio/recepcion de paquetes TimerC;
                // Para realizar temporizaciones
                TimerC.StdControl->GenericComm.StdControl;
        }
}
```

Module

Esta sección es la que por lo general, es más extensa y es en la que realmente se programa el comportamiento que se desea realizar en la aplicación. Ésta a su vez, está dividida en tres subsecciones : ***Uses, Provide, Implementation***.

Antes de empezar con el significado de cada una de estas subsecciones vamos a situarlas dentro del fichero, para que se pueda apreciar la estructura del mismo:

```
module MiAplicacionM {
        provides {… }
        uses {… }
        implementation {…}
}
```

La primera subsección, ***provides***, proporciona al lenguaje de programación cuáles son las interfaces que va a proporcionar nuestro componente, en el caso de que nuestro componente sea una aplicación, como ya se vio anteriormente se ha de proporcionar como mínimo la interfaz *StdControl.*

Pero ahora bien, cuando nosotros informamos que vamos a proporcionar una interfaz, estamos diciendo desde un punto de vista orientado a objetos, que vamos a implementar dicha interfaz, por tanto, se deberán de implementar los métodos que obligue a implementar dicha interfaz.

Dentro de la sección ***provides*** la forma de anunciar que se va a proporcionar una interfaz es la siguiente:

```
provides {
        interface interfazqueproporciono;
}
```

La subsección ***uses*** informa al lenguaje de programación que voy a hacer uso de una interfaz, por tanto si hago uso de una interfaz podré realizar llamadas a los métodos de dicha interfaz.

Sin embargo, cuando nosotros informamos que vamos a utilizar una interfaz, se derivan ciertas acciones que tenemos que realizar para el correcto funcionamiento de nuestra aplicación. Primero, si utilizamos una interfaz, obligatoriamente en la sección de ***implementation*** debe de haber un ***wiring*** que conecte dicha interfaz con un componente que la proporcione. Segundo y último, el utilizar una interfaz conlleva implícitamente el tener que **implementar** los eventos que se puedan producir por el hecho de haber utilizado la interfaz.

La forma de indicar que vamos a utilizar una interfaz es la siguiente:

```
uses {
        interface interfacequeutilizo;
 }
```

Veamos ahora la última subsección, quizás la más extensa, es la subsección ***implementation,*** ésta contendrá todos los métodos necesarios para proporcionar el comportamiento deseado a nuestro componente o aplicación.

La estructura de la misma, es similar a la de un programa en lenguaje C pero con sutiles diferencias, aunque más bien que diferencias son añadidos para poder adaptar la programación al estilo de la programación orientada a eventos.

Esta subsección ha de contener como mínimo

- Las variables globales que va a utilizar nuestra aplicación
- Las funciones que tenga que implementar debido a las interfaces que estoy proporcionando
- Los eventos que tenga que implementar debido a las interfaces de las que estoy realizando uso.

Así que, llegados a este punto vamos a profundizar un poquito más acerca de los diferentes tipos de datos que se nos proporcionan, los deferentes tipos de métodos, y algunas palabras reservadas nuevas.

Tipos de datos

Los tipos de datos que se pueden utilizar en *NesC* son todos los que proporciona *C estándar* más unos cuantos más, que en realidad no aportan potencia de cálculo pero son muy útiles para la construcción de paquetes ya que proporcionan al usuario información acerca del número de bits que ocupan y esto es importante a la hora de transmitir información vía radio.

Los tipos adicionales son:

- *uint16_t* que viene a ser un entero sin signo que ocupa 16 bits
- *uint8_t* que viene a ser un entero sin signo que ocupa 8 bits.
- *result_t,* se utiliza para devolver si usa función se ha ejecutado con éxito o no, viene a ser como un booleano pero con los valores SUCCESS y FAIL
- *bool*, es un valor booleano que puede valer *TRUE* o *FALSE*

Como observación, comentar que si es posible la utilización de memoria dinámica pero no es muy recomendable a no ser que sea absolutamente necesaria, para poder utilizarla existe un componente especial denominado *MemAlloc* que realiza una administración de la memoria dinámica.

Como después veremos existen diferentes tipos de funciones o métodos, uno de estos tipos son las denominadas funciones asíncronas, estas funciones no tienen por que realizarse

de forma inmediata y por tanto cuando una de estas funciones utilice una variable global, esto se ha de indicar de forma explicita para poder realizar una exclusión mutua de la memoria y no perder ningún valor y la forma de indicarlo es anteponer la palabra reservada *norace* delante de la declaración de la variable:

norace uint16_t temperatura;

Tipos de Funciones

En *NesC* las funciones pueden ser de tipos muy variados, primero existen las funciones clásicas con su misma semántica que en *C* y la forma de invocarlas es la misma que en *C*.

Ahora bien, existen otros tipos de funciones añadidos, estos son: ***task***, ***event***, ***command*** así que vamos a explicar sus diferencias y la forma de invocarlos.

Las funciones ***command*** o comandos son básicamente funciones, que al igual que las clásicas se ejecutan de forma síncrona, es decir que cuando son llamadas se produce su ejecución inmediatamente. La forma de llamar a una de estas funciones es *call interfaz.nombreFuncion*.

Las funciones ***task*** o tareas son funciones que se ejecutan concurrentemente en la aplicación, utilizan la misma filosofía que los *hilos o threads*, básicamente es una función normal que se invoca de la siguiente forma: *post interfaz.nombreTarea* y que **inmediatamente** después de su invocación, continua la ejecución del programa invocador.

Las funciones ***event*** o eventos son funciones que son llamadas cuando se levanta una señal en el sistema, básicamente poseen la misma filosofía que la programación orientada a eventos, de manera que cuando el componente recibo un evento se realizará la invocación de dicha función aunque existe un método para poder invocar manualmente este tipo de funciones, es : *signal interfaz.nombreEvento* , además en el caso de tratarse de interfaces parametrizadas poder utilizar la variante *signal interfaz.nombreEvento[parámetro]*.

Ahora bien, estos tres tipos de funciones añadidos, por lo general poseen las características arriba mencionadas pero en su declaración se puede anteponer la palabra reservada ***async*** para indicar que poseen una ejecución asíncrona, este tipo de ejecución por lo general viene dado cuando la ejecución de uno de estos tipos de función viene determinada por el levantamiento de una señal hardware. Por ejemplo, que la temperatura ya se encuentre preparada para ser leída. En el caso de llamar a una de estas funciones asíncronas si se intenta acceder a una variable global, esta se ha de haber declarado con ***norace*** para indicar si exclusión mutua.

Debido a que se permite cierto tipo de programación recurrente mediante la invocación de tareas o *task* el lenguaje de programación permite construir una agrupación de sentencias que conformen una transacción, al estilo de bases de datos, para asegurar así que mientras que se este realizando dicha transacción, las variables implicadas no van a ser modificadas por otro hilo de ejecución.

Esto se consigue mediante la palabra reservada ***atomic*** y la forma de usarlo es la siguiente:

atomic{ … //Sentencias; }

Debido a que ciertas interfaces parametrizadas, no pueden ser llamadas dos veces con el mismo parámetro, existe una función *built-in* que proporciona un valor único para una constante, es decir si llamo a la función muchas veces con el mismo parámetro esta asegura que cada vez va a devolver un valor distinto. Dicha función es *uint_16 unique(string parámetro);*

Componentes Primitivos de TinyOS

TinyOS ofrece al programador innumerables componentes primitivas, como comentarlas todas haría que el documento se transformase en un libro, vamos a ver las más relevantes. La forma de verlas será una simbiosis entre los componentes que proporciona TinyOS y las interfaces que proporcionan, así se podrá tener una idea más general de la finalidad del componente.

Componente Main

Éste es un componente especial que representa el cuerpo main al estilo de C de un programa y que necesita una interfaz StdControl para su funcionamiento, dicha interfaz será proporcionada por el componente que quiera convertirse en una aplicación.

Interfaz StdControl

La interfaz StdControl es una interfaz especial que han de proporcionar todos los componentes que se quieran llamar aplicación y por tanto sean ejecutables en un sensor.
Dicha interfaz obliga a implementar los siguientes métodos

command result_t init();
Este método se va a invocar cuando el sensor se arranque, es decir cuando se enchufe

command result_t start();
Este método se va a invocar después de la invocación del método init() y cuando el sensor pase de off/on

command result_t stop();
Este método se va a invocar cuando el sensor pasa de on/off

Componentes GenericComm y GenericCommPromiscuous

Este componente proporciona mecanismos para poder enviar y recibir paquetes vía radio y vía puerto serie. Actúa como switch entre la radio y el puerto serie, de manera que si un paquete se envía vía radio a la dirección del puerto serie, cuando lo reciba este componente será enviado dicho paquete por el puerto serie.

Utiliza el modelo AM estándar de TinyOS por lo que se basa en el envío de paquetes TOSMsg, estos paquetes poseen la siguiente estructura:

```
uint16_t addr;
uint8_t type;
uint8_t group;
uint8_t length;
int8_t data[TOSH_DATA_LENGTH];
uint16_t crc;
```

El significado de cada uno de los campos es el siguiente:

ADDR es la dirección a la que va destinado el paquete, existen ciertas direcciones especiales:

TOS_BCAST_ADDR es la dirección de broadcast
TOS_LOCAL_ADDRESS es la dirección local (localhost)
TOS_UART_ADDR es la dirección del puerto serie

TYPE es el tipo de paquete que se está enviando, este tipo se utiliza en las interfaces paramétricas para determinar qué instancia de la interfaz se va a hacer cargo de procesar dicho paquete.

GROUP es el grupo al que pertenece el sensor que esta enviando el paquete, de manera que sólo lo reciban aquellos sensores que pertenezcan al mismo grupo, de esta manera pueden coexistir dos redes diferentes de sensores sobre el mismo medio físico aéreo y sobre el mismo canal de radio.

LENGTH es la longitud total del paquete

CRC es la corrección de errores

DATA es el campo a mi parecer más importante y es el que nos posibilita jugar con los paquetes, en este campo se especifican los datos que se van a enviar, estos datos pueden ser una estructura que conforme un nuevo tipo de paquete de nivel superior.

Este componente proporciona las siguientes interfaces:

•SendMsg[TIPO_PAQUETE]

Esta interfaz posibilita el poder enviar paquetes a un determinado destino, estos paquetes han de ser del tipo que se especifique en el parámetro de la interfaz.

•ReceiveMsg[TIPO_PAQUETE]

Esta interfaz posibilita el poder recibir paquetes de un tipo determinado, especificado en el parámetro de la interfaz.

Un sensor recibirá un paquete siempre y cuando el paquete sea enviado por un sensor perteneciente al mismo grupo y que además el tipo de paquete coincida con el parámetro de la interfaz y que el destino del paquete o bien sea de broadcast o bien coincida por mi dirección local (en el último caso, si se trata del componente **GenericCommPromiscuos** se recibirá siempre aunque no sea broadcast, ni vaya dirigido hacia mí).

•StdControl

No olvidar que si un componente proporciona la interfaz StdControl, es por que es una aplicación y por tanto es necesario llamar a los métodos de dicha interfaz a la vez que a los de nuestro componente

Interfaz SendMsg

Esta interfaz proporciona mecanismos para poder enviar mensajes a otros sensores o al puerto serie del ordenador.

Obliga a implementar los siguientes métodos:

command result_t send(uint16_t address, uint8_t length, TOS_MsgPtr msg)

Este método realiza el envío de un mensaje a la dirección address, dicho mensaje es msg (TOS_MsgPtr es un puntero a una estructura TOS_Msg que a modo informativo deberá de ser global ya que si se declara local, la duración de ese mensaje solo será hasta que la función termine y cuando esto ocurre todavía no se ha enviado el mensaje).

El componente que utilice esta interfaz deberá de implementar los siguientes eventos:

event result_t sendDone(TOS_MsgPtr msg, result_t success);

Este evento se va a producir en el caso en el que el envío de un mensaje haya sido satisfactorio.

Interfaz ReceiveMsg

Esta interfaz proporciona los mecanismos necesarios para poder recibir un paquete en el componente que la utilice.

No obliga a implementar ningún método, sin embargo cualquier componente que haga uso de dicha interfaz se verá obligado a implementar los siguientes eventos:

event TOS_MsgPtr receive(TOS_MsgPtr m);

Este evento se va a producir ante la llegada de un paquete y ha de devolver al final de su invocación el mismo paquete que se ha recibido para poder pasarlo a capas de nivel superior.

Componente ADCC

Este componente tiene la misión de ofrecer mecanismos para poder recoger la información de los sensores que posea la placa.

Para ello proporciona dos interfaces:

•ADC[puerto]

Esta interfaz posibilita el poder realizar una conversión analógico-digital de un puerto. Cada uno de los sensores conectados a la placa tiene asignado un puerto, de manera que mediante la especificación del parámetro de la interfaz se decide cual de los sensores es el que se quiere procesar para obtener su valor

•ADCControl[puerto]

Esta interfaz posibilita realizar configuraciones acerca del muestreo de las muestras capturadas por el componente, de manera que se puede configurar su velocidad de muestreo o realizar un remaping de los puertos, para establecer una correspondencia entre los puertos software (comunes a todos los modelos de sensores) y los puertos hardware dependientes del tipo de placa sobre el que se vaya a ejecutar la aplicación

Interfaz ADC

Esta interfaz proporciona los mecanismos necesarios para poder obtener el valor de un sensor en el componente que la utilice.

Obliga a implementar los siguientes métodos:

async command result_t getData();

Este método produce que se empiece un muestreo del valor que posee el sensor, cuando dicho muestreo se haya finalizado, se producirá el evento dataReady()

async command result_t getContinuousData();

Es muy parecido al anterior, pero no produce un único muestreo sino una secuencia continua de ellos (se similar a tener un timer que invoque al método getData() cada cierto tiempo).

Ahora bien, si un componente utiliza esta interfaz esta obligado a implementar los siguientes eventos:

async event result_t dataReady(uint16_t data);

Este evento se producirá cuando se haya realizado un muestreo y el valor del mismo esta en la variable data pero como mención especial decir que es un evento asíncrono por lo cual si deseo almacenar dicha variable necesitaré declarar la variable donde voy a almacenar el valor con la palabra reservada norace

Componente TimerC

Este componente proporciona los mecanismos necesarios para poder llevar a cabo funciones de temporización de hasta 10 timers diferentes y para ello proporciona las siguientes interfaces:

•Timer[id]

Esta interfaz posibilita el poder tener múltiples timers diferentes, y para ello se utiliza su parametrización de manera que como máximo puedo tener 10 timers. Para hacer el wiring de esta interfaz se suele utilizar la función que proporciona el compilador built-in que es unique, la cual proporcionaba un identificador único.

•StdControl

Lo que significa que en la interfaz StdControl que proporcione el componente que estamos desarrollando tendremos que incluir llamadas a las correspondientes funciones de esta interfaz para que la aplicación TimerC funcione correctamente.

Interfaz Timer

Esta interfaz proporciona mecanismos de temporización, para ello obliga a implementar los siguientes métodos:

command result_t start(char type, uint32_t interval);

Este método produce la inicialización o el arranque de la temporización, los parámetros que recibe son *type* para determinar si es una temporización continua, es decir que se repite de forma indefinida cada *interval* segundos, que en caso de ser así tomará el valor *TIMER_REPEAT,* o por el contrario si es un timer eventual que solo se invocará una vez, cuando se halla transcurrido el tiempo, que en este caso el caso será *TIMER_ONE_SHOT.*

command result_t stop();

Este método produce la parada de la temporización del reloj.

Ahora bien, un componente que haga uso de esta interfaz, está obligado a implementar los siguiente eventos:

event result_t fired();

Este evento se producirá cuando se haya pasado el tiempo y por tanto haya terminado el timer, es decir es la acción que se va a ejecutar cada *interval* tiempo.

Estructura de TinyOS

La estructura de los directorios que componen TinyOS es la siguiente:

Directorio	Descripción
/tos/interfaces	Contiene todas las interfaces que son proporcionadas por los componentes primitivos y por las aplicaciones de ejemplo
/tos/lib	Contiene librerías para resolver determinados problemas
/tos/system	Contiene todos los componentes primitivas que proporciona TinyOS
/tos/types	Contiene los tipos que se utilizan en las primitivas de TinyOS
/tos/platform	Contiene los ficheros necesarios para la ejecución en las diversas plataformas
/tos/senseorboard	Contiene los ficheros que son específicos de cada placa.

Aplicaciones TinyOS

Entorno de simulación TOSSIM

Una vez que hemos terminado de realizar la codificación de nuestra aplicación o componente lo que vamos a realizar es la compilación, para ello se utilizan las reglas de make que se encuentran en el directorio /apps/ y además se ha de especificar un nuevo make para nuestra aplicación o componente, dicho make poseerá como mínimo:

PLATFORMS= //plataformas

Esta variable informa al compilador de las posibles placas o tipos de sensores para los que se puede realizar la compilación, en el caso de querer simularla en el ordenador se ha de incluir pc.

COMPONENT= //nombre componente

Esta variable es de estancia obligatoria e informa al compilador del nombre del componente que se desea compilar.

PFLAGS= // directorio de librerias

Esta variable informa al compilador de donde se pueden encontrar los componentes que no son primitivos proporcionados por TinyOS y que la aplicación utiliza, es lo que comúnmente se denomina aplicaciones de terceros o librerías.

SENSORBOARD=telosb

Esta variable informa al compilador del tipo de placa para el que se desea compilar la aplicación, dependiendo del tipo de placa, esta poseerá una serie de sensores o no.

Veamos un ejemplo completo del make para una aplicación que se llama componente.nc

PLATFORMS=mica mica2 mica2dot micaz pc COMPONENT=componente SENSORBOARD=micasb PFLAGS= -I%T/lib/Route include ../Makerules

Bien, en el caso de querer realizar una simulación de la aplicación se ha de compilar para pc invocando al make por el parámetro pc, y con esto obtendremos un fichero ejecutable que corresponderá al fichero que emula un sensor que contiene programado nuestra aplicación.

Ahora bien, para invocar a este ejecutable, cuya estructura interna es un bucle infinito y por tanto nunca va a parar a no ser que abortemos o finalicemos la ejecución del programa, podemos utilizar una seria de parámetros de los cuales los principales son:

•nodbgout Lo que hará es que no mostrara la información de debug por la salida estándar, es decir por consola.

•gui Este parámetro produce que los sensores simuladores esperen a la aplicación grafica de simulación para empezar su ejecución. Esta aplicación es TinyViz

En definitiva la forma de invocación es:

./nombreEjecutable –parametros nºsensores

Donde el número de sensores, es el número que sensores que se desean simular simultáneamente.

A este entorno de ejecución se le denomina TOSSIM y aparte de proporcionar las funciones de simulación, proporciona otra serie de funcionalidades que veremos a continuación.

En cualquier parte de la subsección de implementation de la sección de module se puede incluir la función dbg que tiene como misión mostrar por la salida estándar o por la interfaz grafica de simulación , una cadena que se pase como parámetro, su estilo de pasar parámetros es exactamente igual al que utiliza printf para C estándar.

Dbg(NIVEL_DEBUG ,”cadena %i”,i);

Donde NIVEL_DEBUG es una constante que puede ser definida para poder estabelcer diferentes niveles de debugging (pensar en el modo verbose), por lo general nosotros utilizaremos niveles de debug que están definidos para el usuario, estos son DBG_USR1 y DBG_USR2.

Otra de las funcionalidades que ofrece TOSSIM es la posibilidad de comunicar paquetes entre el pc y el puerto com o el radio del sensor, y para ello proporciona dos puertos de comunicaciones virtuales que corresponden a tossim-uart y tossim-radio, por tanto esto en simbiosis con la utilidad SerialForwared posibilita que una aplicación mande directamente mensajes por radio o bien por el puerto serie de un sensor determinado. Esta utilidad será explicada más adelante.

TinyViz

Es una utilidad que funciona en conjunción con el entorno de simulación TOSSIM y que proporciona una interfaz grafica de usuario para permitir obtener información e interactuar con la simulación.

Esta utilidad se encuentra ubicada en **/tos/tools/java/net/tinyos/sim/** e inicialmente cuando se arranca, en su ventana principal ofrece un botón para conectarse a la simulación (que ha de haber sido ejecutada en segundo plano (&)) y además se le ha de haber indicado el parámetro –gui para que la simulación espere a que se conecte dicha interfaz grafica.

Pero antes de conectarse a la simulación, se han de activar las funcionalidades que se desean en la interfaz grafica, estas funcionalidades se encuentran en el menú Plug-ins y la más relevante es la que posibilita el entorno de depuración, es concretamente debug message, si se habilita esta opción, se puede seleccionar una pestaña en la que cuando se conecte la interfaz a la simulación irán apareciendo todos los mensajes de debug que se hayan especificado en la implementación del componente, pero como la cantidad de los mismos es muy grande, se suele rellenar el campo match con una cadena, de manera que solo se imprimirán en la lista de mensajes, aquellos mensajes que hagan matching con dicha cadena.

Además dicha opción, también permite ver en modo raw los mensajes que se envían entre las diferentes simulaciones de los sensores, para poder comprobar que la comunicación se hace correctamente.

Existen muchos más plug-ins y algunos de ellos muy útiles pero por motivos de extensión no se van a comentar.

Como mención especial, hay que decir que cuando se arrancar esta aplicación, implícitamente se arranca también la aplicación SerialForwared aunque no se aprecie la aparición de la interfaz de usuario de esta aplicación, ya que se arranca de modo silencioso

SerialForwarder

Esta utilidad tiene la misión de ligar un puerto de comunicaciones como el com, tossim-radio, tossim-uart con un puerto TCP del ordenador, de manera que sirve de pasarela entre los dos, es una aplicación muy útil que permite realizar una aplicación de pc que permita enviar por TCP datos, y estos datos serán recibidos por el SerialForwarder y reenviados al puerto que se le haya especificado y viceversa es decir todos los paquete que sean enviados por el sensor al puerto que esta escuchando el SerialForwarder serán retransmitidos con todas las conexiones activas que haya en el puerto TCP en el que se encuentra escuchando.

Por defecto, posee el puerto COM1 y el puerto TCP 9001 pero se pueden cambiar en cualquier momento.

Esta aplicación se encuentra en **/tools/java/net/tinyos/sf/** y su invocación se hace mediante la maquina virtual de java, como nota especial decir que para su correcta invocación, se ha de invocar a la aplicación desde el directorio **/tools/java** ya que los otros directorios existen por que son un mapping del sistema de paquetes que se ha utilizado en la programación de la utilidad

Listen

Esta utilidad tiene la misión de mostrar por consola todos los paquetes que se reciban por un determinado puerto TCP. Es bastante útil, cuando se quiere comprobar que es lo que va a recibir una aplicación del pc.

Esta utilidad se encuentra en el directorio **/tools/java/net/tinyos/tools/.**

Pruebas realizadas en CIMNE

Modificación del código de la aplicación embebida *blink*

La primera prueba de programación realizada con el kit de desarrollo con el que se dispone ha sido algo tan básico como modificar el código de una aplicación existente. Para ello se han necesitado los siguientes componentes:

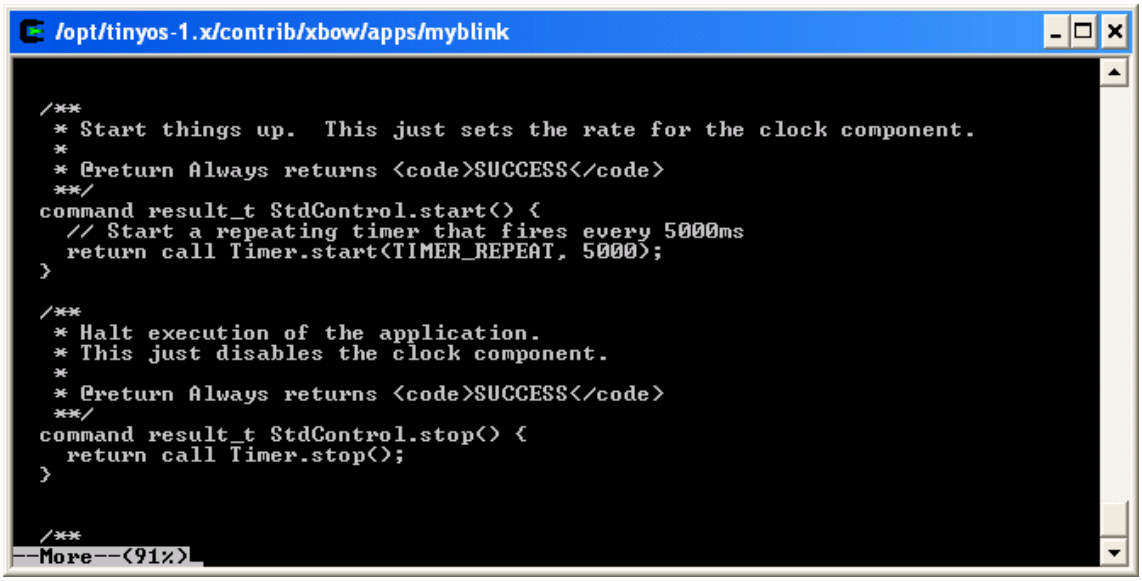
- 1 tarjeta MIB510
- 1 mota MICA2
- 1 cable RS232 con adaptador USB
- 1 PC
- Software: Cygwin
- 1 editor de textos: NotePad

El objetivo de esta primera prueba ha sido familiarizarse con la sintaxis de programación de nesC, así como de las estructuras de los componentes utilizados en ella (configuration, implementation y module). También ha servido para tener un primer contacto con el contenido de los makefiles, necesarios para la compilación de la aplicación.

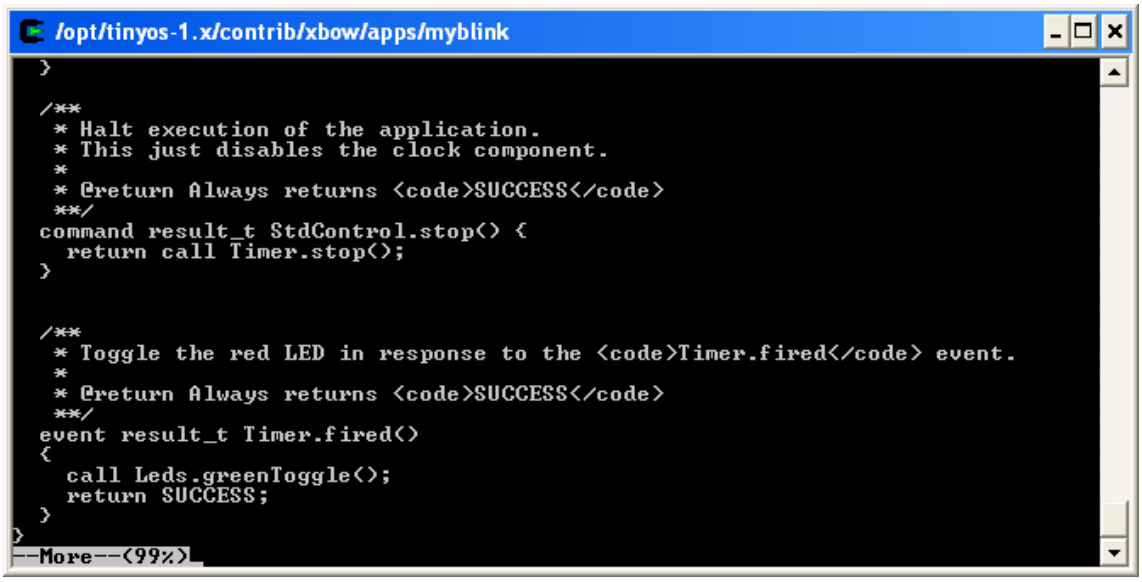
En esta primera práctica hemos querido modificar el código de la aplicación *blink*, que fue la primera aplicación que cargamos en una mota, tal como se explicó en el informe *Plataforma WSN/NIMS II – Primeras experiencias en CIMNE*. Esta aplicación, una vez cargada en la mota hace que parpadee su led rojo, pues bien, hemos querido modificarla de tal manera que sea el led verde el que parpadeara y, además, cambiar el tiempo de frecuencia del parpadeo. Algo muy simple de realizar, pero fundamental para empezar a programar las motas.

Con esta primera experiencia hemos comprendido el uso de dos componentes básicos como son LedsC y TimerC. El primer componente permite controlar el hardware de los leds que contiene la mica, mientras que el segundo nos permite realizar funciones de temporización, necesarias por ejemplo para controlar la frecuencia del parpadeo de los leds o para controlar la frecuencia de muestreo de los datos obtenidos por los sensores acoplados a las motas, algo muy interesante a estudiar.

Para conseguir que el led parpadee cada 5 segundos basta con programar el evento *start* de la interfaz Timer, parametrizándolo con el valor de 5000 milisegundos, tal como muestra la siguiente imagen.



De esta manera el componente TimerC lanzará cada 5 segundos llamadas al evento *fired()* con lo cual basta con programar que este evento haga encender y apagar el led que queramos, en nuestro caso el verde. Para ello aparece la interfaz Leds, que ofrece funciones como *greenToggle()* que nos facilita esta operación.



Programar los leds para que funcionen como un contador en binario

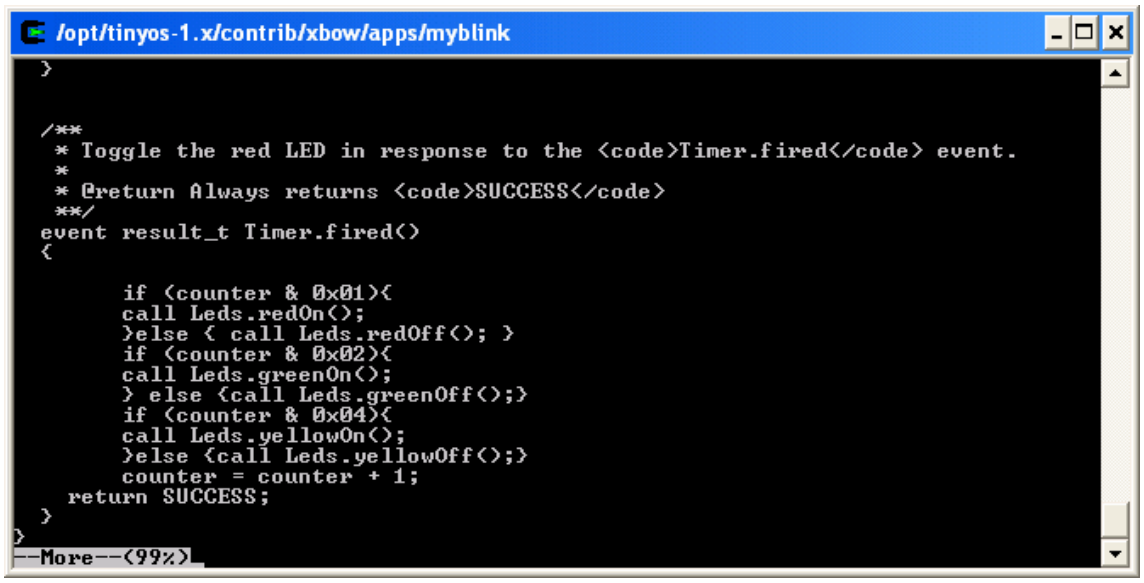
En esta segunda prueba se han utilizado los mismos componentes que en la anterior:

- 1 tarjeta MIB510
- 1 mota MICA2
- 1 cable RS232 con adaptador USB
- 1 PC
- Software: Cygwin
- 1 editor de textos: NotePad

El objetivo de ésta ha sido ampliar la primera práctica, para seguir profundizando en el uso de los componentes TimerC y LedsC. Con ella se ha conseguido reflejar los tres últimos bits de un contador en los leds disponibles de la mota, obteniendo así una especie de contador en binario, que muestra los siguientes ocho estados posibles:

Contador	Led Amarillo	Led Verde	Led Rojo
000	Apagado	Apagado	Apagado
001	Apagado	Apagado	Encendido
010	Apagado	Encendido	Apagado
011	Apagado	Encendido	Encendido
100	Encendido	Apagado	Apagado
101	Encendido	Apagado	Encendido
110	Encendido	Encendido	Apagado
111	Encendido	Encendido	Encendido

Para conseguir este funcionamiento basta con programar el evento *fired()* de un Timer que se ejecuta cada segundo. Así, cada segundo, obtenemos mediante operaciones lógicas, los tres últimos bits de un contador, que es una variable global, *counter*, definida en la aplicación y dependiendo del valor de cada uno de estos tres bits encendemos su correspondiente led, haciendo uso de las correspondientes funciones proporcionadas por el componente LedsC. Este código lo podemos ver reflejado en la siguiente pantalla:



Obtener datos de un sensor y actuar según un rango establecido

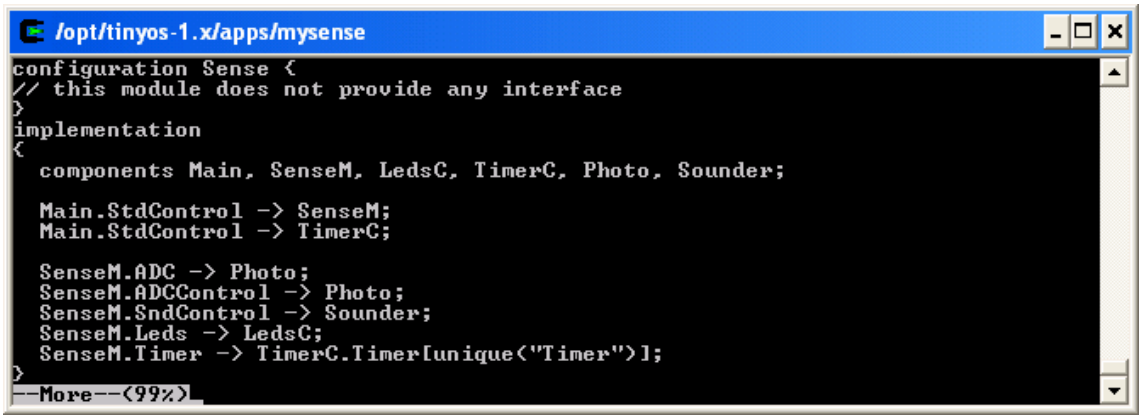
Los componentes utilizados para lograr el correcto funcionamiento de esta tercera prueba han sido los siguientes:

- 1 tarjeta MIB510
- 1 mota MICA2
- 1 tarjeta MTS310
- 1 cable RS232 con adaptador USB
- 1 PC
- Software: Cygwin
- 1 editor de textos: NotePad

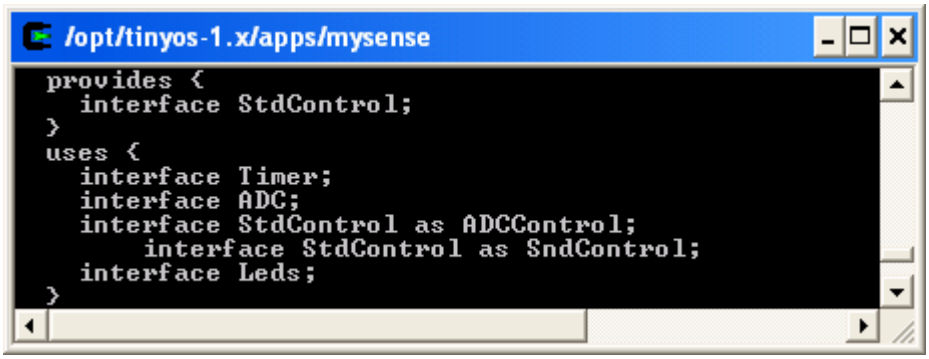
El objetivo principal de esta práctica ha sido lograr obtener por primera vez los datos medidos por un sensor y tratarlos de alguna manera, por ejemplo, capturar el nivel de iluminación de una habitación y actuar según el rango del valor, de manera que si en la habitación se apagan las luces seamos capaces de activar el *sounder* o zumbador que contiene la tarjeta MTS310 para conseguir emitir un pitido que advierta de la baja iluminancia de la habitación que se está monitoreando.

Para lograrlo se han utilizado dos nuevos componentes, aún no vistos, Photo y Sounder, que son los componentes que proporcionan las funcionalidades necesarias para controlar el sensor de luz y el zumbador, respectivamente. Además se ha utilizado la interfaz ADC, que es la encargada de ofrecer los mecanismos necesarios para poder obtener el valor de un sensor en el componente que la utilice.

En la siguiente imagen podemos apreciar la configuración de nuestra aplicación, donde se pueden observar los componentes utilizados, así como la interconexión de interfaces.

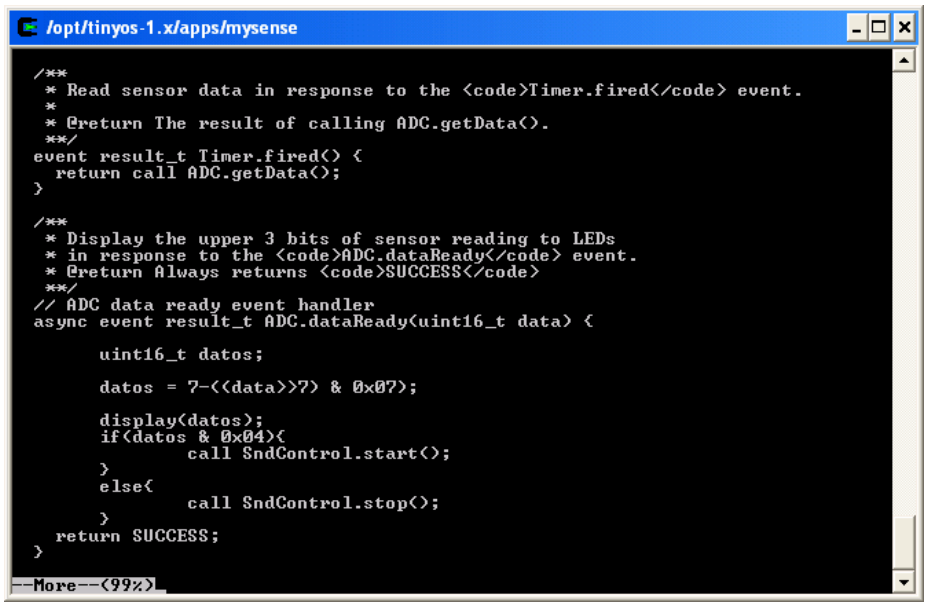


Y en la siguiente captura de pantalla, donde podemos ver parte del módulo de nuestra aplicación, podemos observar las interfaces que usa y provee nuestra aplicación.



Uno de los métodos a recalcar de la interfaz ADC es el *getData()*, que produce que se empiece un muestreo del valor que posee el sensor y cuando dicho muestreo se haya finalizado, se produce el evento *dataReady()*.

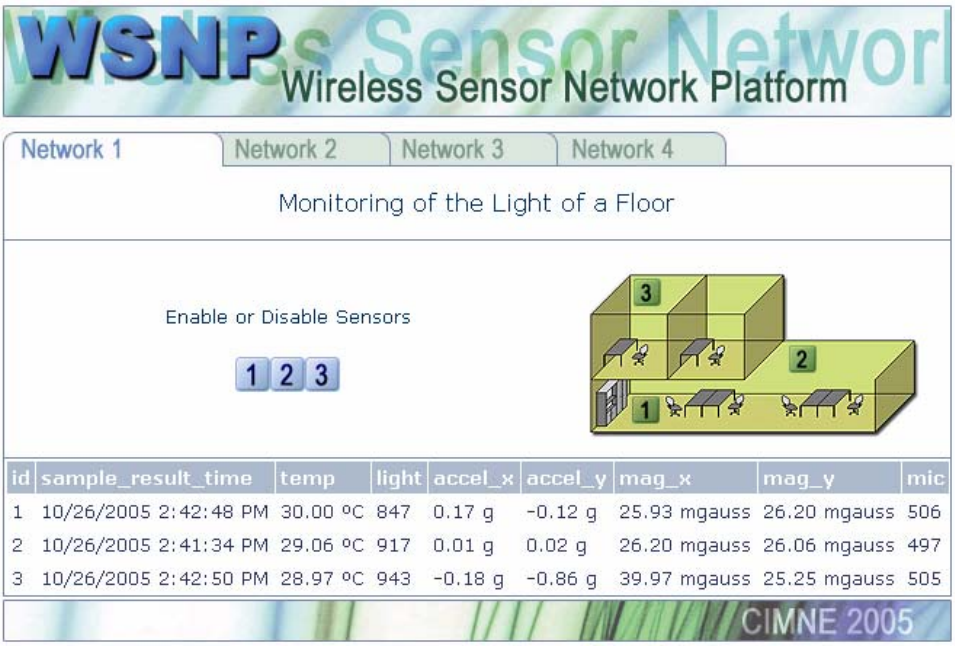
Este evento, *dataReady(uint16_t data)*, se produce cuando se ha realizado el muestreo y su valor está en la variable *data*. Por lo tanto, basta con programar la lógica necesaria en este evento, de manera que recojamos la información guardada en *data*, que es el valor medido por el sensor de luz, y lo comparamos con un rango establecido (previamente estudiado) para decidir si la luz tomada es suficiente para considerar que la luz de la habitación está encendida o apagada. En esta última opción lo que hacemos es actuar sobre el zumbador de la tarjeta MTS310 para que emita un pitido cada vez que detecta las luces apagadas de la habitación.



De esta manera, con esta programación hemos sido capaces, por primera vez, de poder actuar dependiendo de parámetros obtenidos por la medición de determinados sensores.

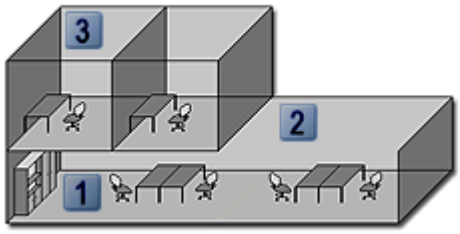
WSNP- Plataforma web capaz de monitorear una red de sensores

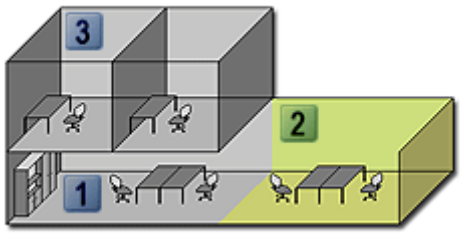
En paralelo a todas las pruebas anteriormente comentadas se ha ido trabajando en el desarrollo de una aplicación web, *Wireless Sensor Network Platform,* capaz, en un primer paso, de monitorear una o varias redes de sensores inalámbricos y con el objetivo de conseguir, a medio/largo plazo, una plataforma más potente que permita, además del seguimiento y mantenimiento de la red, la posibilidad de interactuar con los sensores, abriendo paso a una comunicación bidireccional entre red y plataforma.

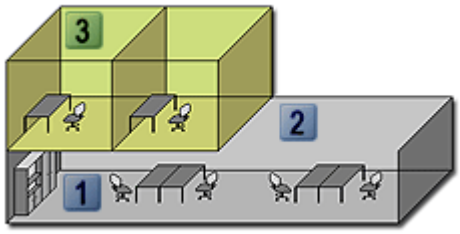


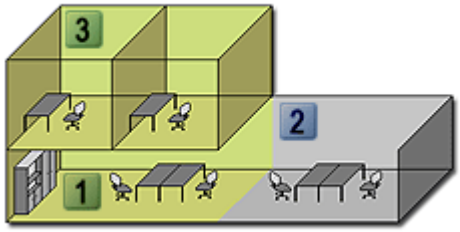
Con esta plataforma se podrá, por ejemplo, reprogramar la red de sensores, cambiar parámetros y funcionalidades de determinados nodos, etc. y todo ello a distancia, sin la necesidad de recoger el gran número de sensores y motas, que pueden llevar a formar una compleja red y volverlos a conectar al puerto serie de un PC para volverlos a programar, lo que supondría, en muchos casos un alto coste de mantenimiento y actualización.

Como primer paso, se ha construido una aplicación web que monitorea a distancia una pequeña red de motas mica2, instalada en varias oficinas ubicadas en dos plantas consecutivas de nuestro centro, con el objetivo de detectar qué habitaciones tienen la luz encendida y cuáles no. Mostrando un mapa dinámico de las plantas, en el que se puede observar cómo se encienden y/o apagan las luces, tal como se muestra en las siguientes figuras.









Además de dicho mapa, también mostramos en una tabla los valores de todos los datos obtenidos por la gama de sensores que comprende una tarjeta MTS310, por lo que tenemos información adicional como pueden ser las temperaturas de las habitaciones, etc. información que, si se desea, también puede ser referenciada gráficamente en el mapa, ya sea con la elaboración de mapas temáticos basado en rango de colores.

id	sample_result_time	temp	light	accel_x	accel_y	mag_x	mag_y	mic
1	10/26/2005 2:42:48 PM	30.00 ºC	847	0.17 g	-0.12 g	25.93 mgauss	26.20 mgauss	506
2	10/26/2005 2:41:34 PM	29.06 ºC	917	0.01 g	0.02 g	26.20 mgauss	26.06 mgauss	497
3	10/26/2005 2:42:50 PM	28.97 ºC	943	-0.18 g	-0.86 g	39.97 mgauss	25.25 mgauss	505

Como se puede observar en este primer prototipo de la plataforma, podemos tener monitoreadas varias redes en paralelo (red de iluminación de una planta, red de control de riego de un jardín, red de control de temperatura de un laboratorio, etc.) incluso se podrían llegar a tener redes de diferentes tipologías y mediciones en una misma zona, sin que se estorbaran entre ellas.

Programación en desarrollo

Las siguientes pruebas no están aún finalizadas, pero se está trabajando actualmente en ellas, al mismo tiempo que se avanza en el desarrollo de la plataforma WSNP.

Programación y estudio de la comunicación vía radio-frecuencia entre motas

Los componentes utilizados para lograr el correcto funcionamiento de esta aplicación han sido los siguientes:

- 1 tarjeta MIB510
- 2 motas MICA2
- 2 tarjetas MTS310
- 1 cable RS232 con adaptador USB
- 1 PC
- Software: Cygwin
- 1 editor de textos: NotePad

El objetivo principal de esta aplicación es conseguir programar dos motas para que se comuniquen entre ellas vía radio-frecuencia. Con ello estudiaremos la estructura interna del paquete que forma un mensaje, así como los componentes e interfaces necesarios para establecer la comunicación correctamente (ver págs. 8 y 9 del informe).

Lo que queremos conseguir a nivel de programación, es extender la aplicación anteriormente detallada, que consistía en actuar sobre el zumbador de la tarjeta MTS310 si el sensor detectaba que se apagaban las luces, y conseguir que la mota que contiene el sensor no emita el pitido, sino que ésta envíe un mensaje por aire a otra mota receptora, para que sea ésta última la que evalúe el valor de la luz tomada por la mota emisora y dependiendo del rango pite ella misma, la receptora y no la que ha tomado el valor del sensor.

OAP (Over Air Programming) Reprogramar una red de sensores por el aire

El objetivo de esta prueba es aprender a "*inyectar*" programas a una red de motas vía radio-frecuencia, de manera que podamos reprogramarlas por el aire, sin necesidad de tener que desmontar la red y reprogramar mota por mota. Funcionalidad que, como se había comentado anteriormente, se añadirá en un futuro a la plataforma WSNP para poder reprogramar y controlar redes de sensores vía web.

Referencias

Tutorial TinyOS – José María Alcaraz Calero

Crossbow – WSN Seminar