Capítulo 4

Integración de sensores: Aspectos Software

En este capítulo se describe el software que se ha desarrollado para cada aplicación, es decir se explicará el funcionamiento y la razón de los programas (componentes e interfaces) que se han creado en lenguaje nesC para la integración de cada uno de los sensores que conforman el proyecto.

4.1 Sensor de monóxido de carbono: *TGS5042*

4.1.1 Descripción general del programa

La aplicación que se ha programado para el sensor *TGS5042* lee las señales de salida del sensor de monóxido de carbono una vez por minuto a través del convertidor analógico-digital del *Mica2*. Éste convierte los datos en unidades de partes por millón y envía un mensaje con ese dato al *Mica2* que hace de receptor.

También evalúa la media de la concentración de monóxido de carbono y envía mensajes de alarma según el estándar europeo *EN50291*. Este estándar indica:

1) Si el ambiente presenta un nivel de concentración de 30*ppm* de *CO* la alarma no debe activarse hasta al menos 120 minutos,
2) Si presenta un nivel de 50*ppm* la alarma no debe activarse antes de 60 minutos pero debe activarse antes de 90 minutos,
3) Si presenta un nivel de 100*ppm* la alarma no debe activarse antes de 10 minutos pero debe activarse antes de 40 minutos
4) Si presenta un nivel de 300*ppm* la alarma debe activarse en 3 minutos.

Para adaptar esta norma se ha adoptado la solución más restrictiva, es decir, si la media de la concentración es mayor o igual que 30*ppm* y menor que 50*ppm* se envía el mensaje de alarma al *Mica2* receptor en 120 minutos, si es mayor o igual que 50*ppm* y menor que 100*ppm* en 60 minutos, si es mayor o igual que 100*ppm* y menor que 300*ppm* en 10 minutos y si es mayor o igual que 300*ppm* en 3 minutos.

El programa está compuesto por los archivos *Makefile* y *Makefile.component*, la configuración *CO.nc* y el módulo *COM.nc*, la configuración *COSensor.nc* y el módulo *COSensorM.nc* y un archivo auxiliar llamado *sensorboardApp.h*.

La configuración *CO* y el módulo *COM* conforman la parte de alto nivel de la aplicación en la que se programa el tratamiento de los datos y el envío de mensajes. La configuración *COSensor* y el módulo *COSensorM* conforman la parte de bajo nivel en la que se trata la adquisición de esos datos mediante el convertidor analógico-digital.

También está el archivo sensorboardApp.h donde se definen los puertos que se utilizan con el convertidor analógico-digital y la estructura de los mensajes que se envían.

4.1.2 Alto nivel. Configuración *CO* y Módulo *COM*

Como se ha dicho, esta parte del programa se ocupa del tratamiento de datos y del envío de mensajes, tanto los mensajes periódicos como los de alarma.

En la configuración *CO.nc* (ver Figura 1 del Anexo II) se define el cableado con los componentes que se utilizan y las interfaces que los relacionan.

Utiliza los componentes *Main*, *COM*, *COSensor*, y los genéricos *TimerC* de temporización, *LedsC* del funcionamiento de los *leds* y *GenericComm* con alias *Comm* para el envío y recepción de mensajes.

Según el cableado los componentes *COM*, *TimerC* y *GenericComm* proveen la interfaz *StdControl* a *Main* lo que supone que la iniciación, el comienzo y la parada de los componentes *TimerC*, *COM* y *GenericComm* son simultáneas.

Las interfaces *Timer* y *Leds* cablean *COM* con los componentes *TimerC* y *LedsC* de manera que pueden ejecutarse en *COM* los comandos y eventos propios de la temporización y el control de leds. La parte de alto nivel y la de bajo nivel se comunican mediante dos interfaces, *COcontrol*, que es un alias de StdControl para el inicio, comienzo y parada de *COSensor* desde *COM* y *Gas*, alias de la interfaz *ADC* (ver Figura 2 del Anexo II), para ordenar desde *COM* la obtención de datos que está programada en *COSensor*.

La interfaz *ADC* tiene dos comandos, *getData* y *getContinuousData* y el evento *dataReady*. *getData* se utiliza para iniciar la conversión analógica-digital y *dataReady* indica que se ha adquirido un dato como resultado a una llamada a *getData* y devuelve ese dato en una variable entera sin signo de 8 *bits*. *getContinuousData* funciona igual que *getData* pero cuando se recibe el evento *dataReady* vuelve a iniciar la conversión.

Por último dos interfaces *SendMsg* y *SendMinuto* que es un alias de *SendMsg*, permitirán el envío de dos tipos de mensajes desde *COM*.

COM.nc provee la interfaz *StdControl* y usa las interfaces *Timer*, *Leds*, *COControl*, *Gas*, *SendMsg* y *SendMinuto* (ver Figura 4.1).

```
module COM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
    interface StdControl as COControl;
    interface ADC as Gas;
    interface SendMsg;
    interface SendMsg as SendMinuto;
  }
}
```

Figura 4.1 Declaración de las interfaces que provee y usa el módulo *COM*

Al principio de la implementación del módulo *COM* primero se definen todas las variables que son necesarias para el programa (ver Figura 3 del Anexo II), tales como índices, búfer de mensajes, variables auxiliares y vectores necesarios.

Estas variables se explicarán según aparezcan en las implementaciones de los comandos, eventos o tareas.

Puesto que *COM* provee la interfaz *StdControl* debe implementar los comandos propios de esta interfaz, *init*, *start*, y *stop*. En el comando *init* (ver Figura 4 del Anexo II) se inician los *leds* y el componente *COSensor* mediante la interfaz *COControl*.

También se inicializa algunas de las variables que vamos a utilizar como por ejemplo se inicializa a cero los 4 vectores de 3, de 10, de 60 y de 120 elementos en los que se almacenan los datos de concentración de *CO* para evaluar las medias. Además se inicializa la cabecera del campo de datos del paquete de mensajes.

En el comando *start* (ver Figura 4.2) simplemente se inicia un temporizador repetitivo que tendrá un periodo de 1 segundo, es decir, 1024 tics del reloj.

```
command result_t StdControl.start() {
  // Comienza un temporizador repetitivo que se dispara cada 1000ms
  return call Timer.start(TIMER_REPEAT, 1000);
}
```

Figura 4.2 Implementación *start*

En el comando *stop* (ver Figura 4.3) se para el temporizador.

```
command result_t StdControl.stop() {
  return call Timer.stop();
}
```

Figura 4.3 Implementación *stop*

El módulo debe además implementar los eventos de las interfaces que usa. Recibe el evento *fired* (ver Figura 4.4) de la interfaz *Timer* cuando termina el temporizador.

```
event result_t Timer.fired()
{
  call Leds.yellowToggle();
  seg=(seg + 1);
  if (seg == 60) {
  seg=0;

  call COControl.start();
  call Leds.greenToggle();
  call Gas.getData();
  }

  return SUCCESS;
}
```

Figura 4.4 Implementación del evento *fired*

Cada vez que el temporizador se dispara se ordena que la luz amarilla de los *leds* alterne y se incrementa la variable *seg* que cuenta los segundos. Cuando la cuenta llega a 60 segundos se alterna la luz verde y se llama al comando *start* de *COControl* para empezar el funcionamiento de *COSensor*. También se llama al comando *getData* de la interfaz *Gas* que controla el convertidor analógico-digital. Así se obtiene un dato cada minuto.

Como ya se ha visto la interfaz gas proporciona un evento asíncrono que indica que los datos están ya listos, es el evento *dataReady* (ver Figuras 5, 6 y 7 del Anexo II) e indica el dato que se ha leído en formato entero (*uint16_t*).

En este evento está programado el tratamiento del dato desde el valor dado por el convertidor que es un número de 0 a 1024 hasta obtener la concentración en *ppm* (de 0 a 10000). Para ello se calculan primero cuantos voltios se han leído en el convertidor y sabiendo la relación entre voltios y *ppm* se calcula la concentración. Es importante hacer los cálculos en variables de tipo no entero (*float*) para obtener resultados correctos.

Después se almacena el resultado obtenido de los cálculos en la variable *PPM*. Luego se asigna valor 5 a una variable llamada *identif* que sirve para diferenciar unos tipos de mensajes de otros. Así pues, el valor 5 de *identif* indica que es un mensaje de datos periódico.

Se rellenan los datos que se requieren en el paquete de mensaje. En la variable *co* se almacena el dato obtenido de *ppm* de *CO*. En la variable *comed* se almacena un cero y en la variable *ident* se almacena el valor de *identif*. Después se llama a la ejecución de la tarea *SendDataMinuto* que envía los mensajes, pero hay que recordar que las tareas se ejecutan sin preferencia sobre los eventos luego la tarea empieza cuando el evento *dataReady* termine.

A continuación se guarda el dato de la concentración en los cuatro vectores de en el elemento cuyo número es igual a un índice que se incrementa cada vez se obtiene un dato. De esta manera el primer dato se almacena en el elemento 0 y el segundo en el 1 y así seguidamente. Cuando el índice llegue al número de elementos del vector, 3, 10, 60 o 120, el índice volverá a cero de manera que en cada vector se tendrán los últimos 3, 10, 60, o 120 datos de concentración que se obtuvieron.

Con los vectores se calculan las medias de los últimos 3, 10, 60 o 120 valores y se almacenan en las variables *ppmed300*, *ppmed100*, *ppmed50* y *ppmed30* respectivamente. Luego se calcula si la media supera los valores de los distintos rangos.

El algoritmo funciona de manera que evalúa si la media de las últimas 3 medidas supera las 300*ppm*, si no evalúa si la media de las últimas 10 mediadas supera las 100*ppm*, si no evalúa si la media de las últimas 60 medidas supera las 50*ppm* y si no evalúa si la media de las últimas 120 medidas supera las 30*ppm*.

Si una de las condiciones se cumple se enciende el *led* rojo y se le asigna un identificador según el valor de la media, si es mayor de 300*ppm* valor 1, entre 100 y 300*ppm* valor 2, entre 50 y 100*ppm* valor 3 y entre 30 y 50*ppm* valor 4. Se almacenan el dato de concentración en ese instante, el dato de la media y el número de identificador en los datos del paquete de mensaje y se ordena la ejecución de la tarea *SendData* que envía el mensaje de alarma. Si no se cumple ninguna de las condiciones se apaga el *led* rojo.

A continuación se explican las tareas *SendData* y *SendDataMinuto* que se ordenan en el evento *dataReady*. El código de ambas tareas es el mismo (ver Figura 8 del Anexo II).

Cuando se ejecuta la tarea se para la ejecución del componente *COSensor* mediante el comando *stop* de la interfaz *COControl*. Se evalúa si la variable booleana *sending_packet* es cierta o no, si es cierta es que ya se está enviando un mensaje y termina allí, pero si no adjudica valor verdadero a esa variable y luego llama al comando *send* de la interfaz *SendMsg* para enviar el mensaje. Si por algún motivo el envío del mensaje no se produce correctamente asigno a la variable booleana el valor falso para indicar que no se está enviando ningún mensaje.

Cuando los mensajes se han enviado correctamente se produce el evento *sendDone* (ver Figura 4.5) de las interfaces *SendData* y *SendMinuto*, en estos eventos lo único que está programado es que se le asigne a la variable booleana el valor falso como indicativo de que no se está enviando un mensaje y así permitir el envío de otros mensajes.

```
event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {
  atomic sending_packet = FALSE;

  return SUCCESS;
}

event result_t SendMinuto.sendDone(TOS_MsgPtr msg, result_t success) {
  atomic sending_packet = FALSE;

  return SUCCESS;
}
```

Figura 4.5 Implementación de los eventos *SendDone* de las interfaces *SendMsg* y *SendMinuto*

4.1.3 Bajo nivel. Configuración *COSensor* y módulo *COSensorM*

La otra parte de bajo nivel del programa está formada por la configuración *COSensor* y el módulo *COSensorM*. Su cometido es la adquisición de los datos mediante el convertidor analógico-digital.

En la configuración *COSensor.nc* (ver Figura 9 del Anexo II) se define el cableado con los componentes que se utilizan y las interfaces que los relacionan.

Como se observa la configuración *COSensor* provee las interfaces *ADC* de alias *ExternalCOADC*, que se ocupa de la conversión analógico-digital y *StdControl* de alias *COStdControl* a la parte de alto nivel del programa.

Los componentes que utiliza son el de temporización *TimerC*, el de la conversión analógico-digital *ADCC*, y el módulo *COSensorM*. Las interfaces *ExternalCOADC* y *COStdControl* que provee *COSensor* están redirigidas mediante el signo = a *COSensorM*, por tanto es como si las proveyera *COSensorM*.

Por otro lado *COSensorM* usa una serie de interfaces. *InternalCOADC* es un alias de la interfaz *ADC* que usa para ordenar la extracción de datos en el componente *ADCC*. *ADCControl* (ver Figura 4.6) es una interfaz que provee el componente *ADCC*.

```
interface ADCControl {
  command result_t init();

  command result_t setSamplingRate(uint8_t rate);

  command result_t bindPort(uint8_t port, uint8_t adcPort);
}
```

Figura 4.6 Interfaz *ADCControl*

Tiene tres comandos, *init* que inicializa las estructuras de *ADCControl*, *setSamplingRate* con el que se determina la frecuencia de muestreo del puerto analógico-digital y *bindPort* con el que se determina que puerto lee el dato. Requiere dos datos, el número de puerto desde el que se lee el dato y el número de puerto del que quiere modificar el mapeado. Generalmente estos dos números son iguales. La cadena de caracteres *TOS_ADC_CO_PORT* se define en el archivo auxiliar e indica el número del puerto que se utiliza.

También se cablea *COSensorM* con el componente de temporización *TimerC* mediante las interfaces *COTimer*, alias de *Timer* para ordenar el comienzo y la parada de los temporizadores y *TimerControl* que es un alias de la interfaz *StdControl* para iniciar, comenzar y parar *TimerC*.

El módulo *COSensorM* provee las interfaces *COStdControl* y *ExternalCOADC* y usa las interfaces *ADCControl*, *InternalCOADC*, *TimerControl* y *COTimer* (ver Figura 4.7).

```
module COsensorM
{
    provides interface StdControl as COStdControl;
    provides interface ADC as ExternalCOADC;
    uses
    {
        interface ADCControl;
        interface ADC as InternalCOADC;
        interface StdControl as TimerControl;
        interface Timer as COTimer;
    }
}
```

Figura 4.7 Interfaces que provee y usa *COSensorM*

El módulo *COSensorM* debe implementar los comandos de las interfaces que provee, *COStdControl* y *ExternalCOADC*. Así pues, implementa el *init*, *start* y *stop* de *StdControl* (ver Figura 10 del Anexo II).

En *init* primero se selecciona el puerto del convertidor que se utiliza mediante el comando *bindPort* de la interfaz *ADCControl*. *TOS_ACTUAL_CO_PORT* es otra cadena de caracteres definida en el archivo auxiliar y se usa para la modificación del mapeado de los puertos. También se inicia el componente de temporización *TimerC*. Finaliza llamando al inicio de *ADCC* mediante la interfaz *ADCControl*. En los comandos *start* y *stop* no se hace nada.

Los comandos que debe implementar de *ADC* son *getData* y *getContinuousData*. En *getContinuousData* no se implementa nada porque no se le va a llamar en el programa de alto nivel y en *getData* se ordena la obtención de una muestra mediante la tarea *getSample* (ver Figura 4.8).

```
async command result_t ExternalCOADC.getData()
{
    post getSample();
    return SUCCESS;
}

async command result_t ExternalCOADC.getContinuousData()
{
    return SUCCESS;
}
```

Figura 4.8 Implementación de los comandos *getData* y *getContinuousData* de la interfaz *ExternalCOADC*

La tarea *getSample* (ver Figura 4.9) para el temporizador si este está corriendo y luego comienza otro temporizador de un solo disparo y de 10 tics de reloj y si el temporizador falla vuelve a invocar la tarea *getSample*.

```
// Obtiene la siguiente muestra
task void getSample()
{
    call COTimer.stop(); // Sólo si es necesario
    if (call COTimer.start(TIMER_ONE_SHOT, 10) != SUCCESS)
    {
        post getSample();
    };
    return;
}
```

Figura 4.9 Tarea *getSample*

Cuando el temporizador termina se produce el evento *fired* en el que se ordena a *ADCC* la obtención del dato a través del puerto mediante *InternalCOADC.getData()* y si se produce correctamente la obtención del dato se pone al *Mica2* en espera un pequeño tiempo (ver Figura 4.10).

```
// Despues de esperar un poco se toma una lectura
event result_t COTimer.fired()
{
    if (call InternalCOADC.getData() == SUCCESS)
    {
        TOSH_uwait(1000);
        return SUCCESS;
    };
    return SUCCESS;
}
```

Figura 4.10 Implementación del evento *fired* de la interfaz *COTimer*

El componente *ADCC* obtendrá entonces el dato del puerto y enviará el evento *InternalCOADC.dataReady* con el dato que se ha extraído (ver Figura 4.11). Este evento lo que hace es producir la señal *ExternalCOADC.dataReady* que también contiene el dato. Esta señal produce la activación del evento *ExternalCOADC.dataReady*. Este evento no hace realmente nada pero la señal será recibida también por el módulo *COM* en el evento *Gas.dataReady* ya que los componentes están cableados. El evento *Gas.dataReady* es el que recibe el dato y lo procesa y luego evalúa si tiene que hacer saltar la alarma.

```
    default async event result_t ExternalCOADC.dataReady(uint16_t data)
    {
        return SUCCESS;
    }

    async event result_t InternalCOADC.dataReady(uint16_t data)
    {
        return signal ExternalCOADC.dataReady(data);
    }

}
```

Figura 4.11 Implementación de los eventos *dataReady* de las interfaces *InternalCOADC* y *ExternalCOADC*

4.1.4 Archivo Auxiliar *sensorboardApp*

Para que todo el conjunto de los programas esté completo se hace necesario el archivo auxiliar *sensorboardApp.h* (ver Figura 11 del Anexo II). Este archivo contiene definiciones específicas que se utilizan en el programa. Además de estas definiciones se ha incluye también la estructura del campo *data* del mensaje.

En este archivo se encuentra la definición de las cadenas de caracteres TOS_ACTUAL_CO_PORT y TOS_ADC_CO_PORT como valor tres. Esto indica que se utiliza el puerto convertidor analógico-digital 3.

La estructura del campo *data* del mensaje se divide en dos partes, la cabecera y los datos propiamente dichos. En la cabecera se encuentran las variables necesarias para

identificar el mensaje, los identificadores de la placa de adquisición de datos, del paquete, y del nodo que manda el mensaje.

Para esta aplicación la única variable que puede cambiar de un mensaje a otro es el identificador del nodo si hay más de un nodo emitiendo. Este dato permite conocer la procedencia del mensaje cuando los reciba el *Mica2* receptor y el programa *Xsniffer* lo muestre por pantalla.

Como ya se ha visto la cabecera se rellena al inicio del módulo *COM.nc*. La parte que contiene los datos contiene tres variables, la variable *ident* será un número del uno al cinco e indicará el tipo de mensaje que se ha recibido. Si contiene un cinco será un mensaje periódico (no de alarma), si es un número del cuatro al uno significa que ha saltado la alarma porque la media de concentración de *CO* ha superado la concentración límite durante los tiempos establecidos.

Así, si supera las 30ppm en 120 minutos *ident* valdrá cuatro, si supera las 50ppm en 60 minutos valdrá tres, si supera las 100ppm en 10 minutos valdrá dos y si supera las 300ppm en 3 minutos valdrá uno. Esto permite conocer porque razón se ha enviado el mensaje de alarma. Este dato se rellena justo antes de ordenar el envío de mensajes en el módulo *COM.nc*.

La variable *co* contiene el valor de la concentración de *CO* en partes por millón, es el dato que se envía cada minuto. La variable *comed* contiene el dato de la media en el tiempo de la concentración de *CO* en partes por millón. En los mensajes periódicos la variable *comed* vale cero porque solo interesa el valor de concentración en ese instante. Sin embargo en los mensajes de alarma ambas variables *co* y *comed* se rellenan.

También se define la cadena de caracteres *AM_XSXMSG* como cero. Esta cadena como se vio en el capítulo 2 es un argumento de la interfaz *SendMsg* y se almacena el campo *type* de los mensajes de *TinyOS* indicando el tipo de mensaje.

4.2 Sensor de dióxido de carbono: *CDM4161*

4.2.1 Descripción general del programa

La aplicación que se ha programado para el módulo *CDM4161* lee la señal de salida de la concentración de dióxido de carbono cada cinco segundos mediante el convertidor analógico-digital del *Mica2*, procesa ese dato convirtiéndolo en partes por millón de CO_2 y lo envía en un mensaje al *Mica2* que hace de receptor.

La aplicación también vigila el nivel de la otra salida del módulo, la señal de control, mediante un puerto de interrupción. Si la señal de control pasa de nivel bajo a nivel alto el programa generará una interrupción y ésta a su vez provocará el envío de un mensaje al *Mica2* receptor.

Los componentes que se han programado para la aplicación pueden dividirse en tres partes bien diferenciadas. Por un lado está el que se podría llamar programa de más alto nivel formado por la configuración *CO2.nc* y el módulo *CO2M.nc* cuyo cometido es el de recibir los datos y tratarlos, recibir la notificación de que se ha producido una interrupción y enviar mensajes al nodo receptor.

Por debajo de este nivel están las otras dos partes del programa, una comprende la adquisición de datos a través del convertidor analógico-digital y está formada por la configuración *CDM4161.nc* y el módulo *CDM4161M.nc* y la otra se ocupa de la activación y desactivación de las interrupciones y de la notificación de la interrupción y está formada por la configuración *CO2Sensor.nc* y el módulo *CO2SensorM.nc*. Para hacer posible la comunicación de ésta última parte con la parte de alto nivel se ha creado la interfaz *CO2Control.nc*.

4.2.2 Alto nivel. Configuración CO2 y módulo CO2M

La parte de alto nivel del programa está formada por la configuración *CO2.nc* y el módulo *CO2M.nc*. La configuración *CO2.nc* (ver Figura 12 del Anexo II) define los componentes que se utilizan y sus relaciones mediante interfaces.

Los componentes que utiliza además de *Main* son: su respectivo módulo *CO2M*, las configuraciones de las partes de bajo nivel *CDM4161* y *CO2Sensor*, el componente que controla los *leds LedsC*, el que controla la temporización *TimerC*, y Comm alias de *GenericCommPromiscuous* que es una versión de *GenericComm*.

Si se observa el cableado se puede ver que mediante la interfaz *StdControl* el inicio, funcionamiento y parada del programa son simultáneos en los componentes *CO2M*, *LedsC* y *Comm*.

Se observa que mediante las interfaces *Leds* y *Comm* en el componente *CO2M* se controlará el encendido/apagado de los *leds* y el envío de mensajes respectivamente. También, mediante la interfaz *CO2Control* (ver Figura 13 del Anexo II), que se ha creado para controlar la parte del programa dedicada a las interrupciones (*CO2Sensor*).

CO2M utiliza un temporizador mediante la interfaz *Timer1* y también *CO2M* toma el control (inicio, comienzo y parada) mediante la interfaz *CDMControl* de la parte que se dedica a la adquisición de datos del convertidor analógico-digital (*CDM4161*). El último cableado es el de *CO2M* con *CDM4161* a través de la interfaz *CODOS* que es un alias de la interfaz *ADC* que se ocupa de la orden de obtención de un dato de conversión analógica-digital través un puerto y de la notificación de que el dato está listo.

Por tanto el módulo *CO2M* (ver Figura 4.12) provee la interfaz *StdControl* y usa las interfaces *ADC*, *Leds*, *SendMsg*, *Timer1*, *CO2Control* y *CDMControl*.

```
module CO2M {
  provides {
    interface StdControl;
  }
  uses {
    interface ADC as CODOS;
    interface SendMsg;
    interface CO2Control;
    interface StdControl as CDMControl;
    interface Timer as Timer1;
  }
}
```

Figura 4.12 Interfaces que provee y usa el módulo *CO2M*

Al comienzo de la implementación (ver Figura 4.13) se declaran las variables que se utilizan en el programa tales como variables para almacenar datos, buffer, paquete de mensajes y una variable booleana que sirve para distinguir cuando se está enviando un mensaje dependiendo de si su valor es verdadero o falso. Se iniciará en su declaración con valor falso indicando que no se está enviando ningún mensaje.

```
implementation {
  bool sending_packet = FALSE;
  TOS_Msg msg_buffer;
  XDataMsg *pack;
  uint16_t control_signal;
  float co2;
  float co22;
  uint16_t co2ppm;
```

Figura 4.13 Declaración de variables

Después del inicio de variables están implementados los comandos de la variable *StdControl* que provee *CO2M*, *init*, *start* y *stop* (ver Figura 14 del Anexo II).

En el comando *init* se inician los *leds* y se inicia mediante la interfaz *StdControl* la parte de bajo nivel que se ocupa de la extracción de datos del convertidor analógico-digital. Además se rellena el paquete de mensaje con los valores que van por defecto en todos los mensajes.

En el comando *start* se habilitan las interrupciones en *CO2Sensor.nc* mediante una llamada al comando *start* de la interfaz *CO2Control* y se inicia un temporizador repetitivo de cinco segundos (*Timer1*). Cada vez que terminan los cinco segundos se obtendrá un dato del convertidor analógico-digital.

En el comando *stop* se para el temporizador y se deshabilitan las interrupciones en *CO2Sensor* mediante una llamada al comando stop de *CO2Control*.

Los eventos de las interfaces usadas por *CO2M* también deben estar implementados. Cada vez que el temporizador *Timer1* termina la cuenta se produce el evento *Timer1.fired* (ver Figura 4.14).

```
event result_t Timer1.fired()
{
  call CDMControl.start();
  call CODOS.getData();

  return SUCCESS;
}
```

Figura 4.14 Implementación del evento *Timer1.fired*

En el evento se ordena el comienzo de la parte dedicada al convertidor analógico-digital (*CDM4161.nc*) mediante la interfaz *CDMControl* y se ordena la adquisición de datos mediante la interfaz *CODOS*.

Cuando el dato esté listo llegará al programa el evento asíncrono *CODOS.dataready* (ver Figura 4.15) con el dato extraído.

```
async event result_t CODOS.dataReady(uint16_t data) {
  atomic co2 = (float)data;
  atomic co22 = (((co2/1024)*3)*2000);
  atomic co2ppm = (uint16_t)co22;
  atomic pack->xData.datap1.ppm=co2ppm;
  post SendData();
  return SUCCESS;
}
```

Figura 4.15 Implementación del evento *CODOS.dataReady*

En este evento se procesará el dato obtenido. Puesto que el convertidor funciona para tensiones de 0 a 3*V* y mide valores de 0 a 1024 unidades dividiremos el valor obtenido por 1024 y multiplicaremos por 3 para obtener el dato en voltios. Luego multiplicaremos por 2000 puesto que la salida del módulo es directamente proporcional a la concentración de CO_2 en una proporción de 2000*ppm* por voltio. Así pues, obtendremos el dato en partes por millón y luego lo almacenaremos en la variable *co2ppm* del paquete de mensajes.

Se ha de tener cuidado con las variables que se utilizan para no perder información con los cálculos, ya que por ejemplo si dividimos entre 1024 y la variable es entera, como no puede valer más de 1024, el resultado de la división dará 0 y al multiplicar se obtendrán 0*ppm* lo cual no será cierto.

Una vez que la variable está calculada se ordenará el envío de los datos mediante la tarea *SendData* (ver Figura 4.16).

```
/**Tarea que manda mensajes a través de la radio**/
void task SendData(){
  call CDMControl.stop();
  if (sending_packet) return;
    atomic sending_packet = TRUE;
  if (call SendMsg.send(TOS_BCAST_ADDR,sizeof(XDataMsg),&msg_buffer)!= SUCCESS)
    sending_packet = FALSE;
  return;
}
```

Figura 4.16 Tarea *SendData*

En la tarea *SendData* se evaluará la variable booleana que nos indica si se está enviando un mensaje en ese momento y si no es así se le asigna el valor verdadero que indica que

a partir de ese momento si se está enviando. Seguidamente se llama al comando *send* de la interfaz *SendMsg* para enviar por radio mediante *GenericCommPromiscuous*. Si el envío es fallido a la variable booleana se le asigna valor falso. Si esto se produce correctamente se genera el evento *SendDone* (ver Figura 4.17).

```
/**Evento que avisa que se ha enviado un mensaje correctamente**/
event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {
  atomic sending_packet = FALSE;

  return SUCCESS;
}
```

Figura 4.17 Implementación del evento *SendDone* de la interfaz *SendMsg*

En él lo que se hace es asignar valor falso a la variable booleana para indicar que ya no se está enviando un mensaje.

Por otro lado al haber habilitado las interrupciones se puede recibir una interrupción que generará el evento *CO2Control.fired* (ver Figura 4.18).

```
/**Evento que indica una interrupción, es decir se ha detectado fuego**/
/**Asigna un 1 al apartado fire del mensaje**/
event result_t CO2Control.fired(){
atomic control_signal = 1000;
atomic pack->xData.datap1.control = control_signal;
atomic pack->xData.datap1.ppm=0;
post SendCO2();
return SUCCESS;
}
```

Figura 4.18 Implementación del evento *fired* de la interfaz *CO2Control*

En él se asignará valor 1000 a una variable (el valor del umbral por defecto para que salte la señal de control) y se almacena en el paquete de mensajes en la variable *control*. A la variable *ppm* del paquete se le asigna valor cero. Seguidamente se ordena el envío del mensaje mediante la tarea *SendCO2* (ver Figura 4.19) que es igual que la tarea *SendData* que se vio antes. Si se produce correctamente el envío también se generará el evento *SendDone*.

```
/**Tarea que manda mensajes a través de la radio**/
void task SendCO2(){
  call CDMControl.stop();
  if (sending_packet) return;
    atomic sending_packet = TRUE;
  if (call SendMsg.send(TOS_BCAST_ADDR,sizeof(XDataMsg),&msg_buffer)!= SUCCESS)
    sending_packet = FALSE;
  return;
}
```

Figura 4.19 Tarea *SendCO2*

4.2.3 Bajo nivel. Configuración *CO2Sensor* y módulo *CO2SensorM*

Ésta parte del programa se ocupa de las interrupciones. La interfaz *CO2Sensor* es la única que provee la configuración *CO2Sensor.nc* (ver Figura 15 del Anexo II) y el único componente es el módulo *CO2SensorM*.

El cableado *CO2Control=CO2SensorM* indica que todo lo que esté cableado con *CO2Sensor* mediante *CO2Control* en realidad está cableado con *CO2SensorM* (ver Figura 4.20).

```
module CO2SensorM {
  provides {
    interface CO2Control;
  }
}
```

Figura 4.20 Interfaz que provee *CO2SensorM*

En el módulo *COSensorM* están programados los comandos *start* (ver Figura 4.21) y *stop* (ver Figura 4.22) de la interfaz que provee (*COControl*) y el proceso que tiene lugar al detectar una interrupción.

```
/**Habilita las interrupciones**/
command result_t CO2Control.start() {
  TOSH_CLR_CO22_PIN();          // Inicializa a cero el pin conectado al sensor
  TOSH_MAKE_CO22_INPUT(); // Convierte entrada el pin conectado al sensor
  sbi(EICRB,ISC51);             // Interrupción por flanco de subida
  sbi(EICRB,ISC50);
  sbi(EIMSK,5);                 // Habilita las interrupciones en INT1
  return SUCCESS;
}
```

Figura 4.21 Implementación del comando *start* de la interfaz *CO2Control*

En el comando *start* se pone a cero el pin que va a ser utilizado para detectar la interrupción mediante la orden *TOSH_CLR_CO22_PIN()* y luego se selecciona como entrada mediante la orden *TOSH_MAKE_CO22_INPUT()* ya que ese pin puede ser tanto entrada como salida.

Después para seleccionar la interrupción por flanco de subida se ponen a nivel alto mediante el comando *sbi* los registros *ISC51* y *ISC50* del control de interrupciones externas *B* y finalmente para habilitar las interrupciones se pone a nivel alto la máscara de interrupciones externas para el pin *INT1* que es el que se utiliza.

```
/**Deshabilita las interrupciones**/
command result_t CO2Control.stop() {
  cbi(EIMSK,5);                 // Deshabilita las interrupciones en INT1
  return SUCCESS;
}
```

Figura 4.22 Implementación del comando *stop* de la interfaz *CO2Control*

En el comando stop se deshabilitan las interrupciones poniendo a nivel bajo la máscara de interrupciones para INT1.

Cuando llega una interrupción del puerto *INT1* se ejecuta el evento *TOSH_SIGNAL(SIG_INTERRUPT5)* (ver Figura 4.23).

```
/**Se ejecuta en caso de interrupción*/
 TOSH_SIGNAL(SIG_INTERRUPT5)
{
  cbi(EIMSK,5);                    // Deshabilita interrupciones
  signal CO2Control.fired(); // Señala que se ha producido una interrupción
  sbi(EIMSK,5);                    // Habilita interrupciones
}
```

Figura 4.23 Implementación en caso de recepción de una señal de interrupción

En la implementación se deshabilitan las interrupciones del *INT1*, se manda la señal *CO2Control.fired* al módulo *CO2M.nc* y luego se vuelve a deshabilitar las interrupciones.

4.2.4 Bajo nivel. Configuración *CDM4161* y módulo *CDM4161M*

La otra parte de bajo nivel del programa es la que se dedica a la obtención del dato a través del convertidor analógico-digital y está formada por la configuración *CDM4161.nc* (ver Figura 16 del Anexo II) y por el módulo *CDM4161M.nc*.

Como se puede ver es totalmente análogo al código de la configuración *COSensor* del sensor de *CO* y lo mismo ocurre con el módulo *COSensorM* del sensor de *CO* y el *CDM4161M* (ver Figura 4.24) de éste sensor.

```
module CDM4161M
{
    provides interface StdControl as CDMStdControl;
    provides interface ADC as ExternalCDMADC;
    uses
    {
        interface ADCControl;
        interface ADC as InternalCDMADC;
        interface StdControl as TimerControl;
        interface Timer as CDMTimer;
    }
}
```

Figura 4.24 Interfaces que provee y usa el módulo *CDM4161M*

El módulo *CDM4161M* implementa los comandos de las interfaces que provee, *StdControl* y *ADC*. Así implementa el *init*, *start* y *stop* (ver Figura 17 del Anexo II) de la interfaz *StdControl*.

En *init* primero se selecciona el puerto del convertidor que se va a utilizar y luego se inicia *ADCC* mediante la interfaz *ADCControl*. También se inicia *TimerC*. En los comandos *start* y *stop* no se hace nada.

Los comandos de *ADC* son *getData* y *getContinuousData*. En *getContinuousData* no se hace implementa nada puesto que no va a ser llamada y en *getData* (ver Figura 4.25) se ordena la obtención de una muestra mediante la tarea *getSample*.

```
async command result_t ExternalCDMADC.getData()
{
    post getSample();
    return SUCCESS;
}
```

Figura 4.25 Implementación del comando *getData* de la interfaz *ExternalCOADC*

La tarea *getSample* (ver Figura 4.26) para el temporizador si este está corriendo y luego comienza otro temporizador de un solo disparo y de 10 tics de reloj y si el temporizador falla vuelve a invocar la tarea *getSample*.

```
// Obtiene la siguiente muestra
task void getSample()
{
    call CDMTimer.stop(); // Sólo si es necesario
    if (call CDMTimer.start(TIMER_ONE_SHOT, 10) != SUCCESS)
    {
        post getSample();
    };
    return;
}
```

Figura 4.26 Implementación de la tarea *getSample*

Cuando el temporizador que se inicia en esta tarea termina se produce el evento *fired* (ver Figura 4.27) en el que ordenaremos a *ADCC* la obtención del dato a través del puerto mediante *InternalCDMADC.getData* y si se produce correctamente la obtención del dato se pone al *Mica2* en espera un pequeño tiempo.

```
// Despues de esperar un poco se toma una lectura
event result_t CDMTimer.fired()
{
    if (call InternalCDMADC.getData() == SUCCESS)
    {
        TOSH_uwait(1000);
        return SUCCESS;
    };
    return SUCCESS;
}
```

Figura 4.27 Implementación del evento *fired* de la interfaz *CDMTimer*

Como resultado a la llamada *getData* el componente *ADCC* obtiene entonces el dato del puerto y se produce el evento *InternalCDMADC.dataReady* (ver Figura 4.28) que contiene el dato que se ha extraído. Este evento produce la señal *ExternalCDMADC.dataReady* que también contiene el dato.

```
async event result_t InternalCDMADC.dataReady(uint16_t data)
{
    return signal ExternalCDMADC.dataReady(data);
}
```

Figura 4.28 Implementación del evento *dataReady* de la interfaz *InternalCDMADC*

Esta señal produce el evento *ExternalCDMADC.dataReady* (ver Figura 4.29) que aunque en este módulo no hace realmente nada será recibida e implementada por el módulo *CO2M* en el evento *CODOS.dataReady* ya que los componentes están cableados. El evento *CODOS.dataReady* es el que recibe el dato y lo procesa y ordena el envío.

```
default async event result_t ExternalCDMADC.dataReady(uint16_t data)
{
    return SUCCESS;
}
```

Figura 4.29 Implementación del evento *dataReady* de la interfaz *ExternalCDMADC*

4.2.5 Archivo auxiliar *sensorboardApp.h*

Para completar el programa se necesita, al igual que el programa para el sensor de CO, un archivo auxiliar llamado *sensorboardApp.h* (ver Figura 18 del Anexo II). En este archivo se encuentra la definición del paquete de mensaje, cadenas de mensajes que en realidad son números que utilizaremos en la asignación de puertos para la conversión analógica-digital y la asignación de un alias al pin de interrupción que se utiliza.

Primero se le asigna el alias *CO22* al puerto de interrupción que se utiliza, el puerto *INT1* mediante la orden *TOSH_ALIAS_PIN(CO22, INT1)*. Este alias se utiliza en el módulo *COSensorM* para inicializar el puerto a cero y seleccionarlo como entrada mediante las órdenes *TOSH_CLEAR_CO22_PIN()* y *TOSH_MAKE_CO22_INPUT()*.

Se definen las cadenas de caracteres *TOS_ADC_CO2_PORT* y *TOS_ACTUAL_CO2*_PORT como número 3, lo que indica que se utiliza el puerto *ADC3* para la conversión analógica digital.

La estructura de campo *data* del paquete de mensajes es del mismo tipo que en el programa para el sensor de *CO*. Tiene una cabecera con variables que sirven para identificar el mensaje (el nodo que envía los datos) y una parte de datos formada por las variables *control* y *ppm*.

Según el tipo de mensaje, mensajes de lectura de la conversión analógica-digital o mensajes debidos a las interrupciones por superar un umbral de *CO2* sólo estará relleno un dato. En los mensajes periódicos de lectura de la conversión analógica digital será la variable *ppm* que como su nombre indica contiene el valor de la concentración de CO_2 en partes por millón y en los de interrupción está rellena sólo la variable *control*.

También se define la cadena de caracteres *AM_XSXMSG* como cero. Esta cadena como se vio en el capítulo 2 es un argumento de la interfaz *SendMsg* y se almacena el campo *type* de los mensajes de *TinyOS* indicando el tipo de mensaje.

4.3 Sensor de hidrógeno, metano y gases licuados del petróleo: *FCM6812*

4.3.1 Descripción general del programa

La aplicación que se ha programado para el sensor *FCM6812* lee la señal de salida del módulo a través del convertidor analógico-digital cada cinco segundos y traduce ese dato a partes por millón de cada uno de los gases que puede detectar, hidrógeno, metano, propano e iso-butano. Luego envía un mensaje vía radio con la concentración de los cuatro gases que recibirá un *Mica2* receptor.

El programa está formado por dos partes. La de más alto nivel que se encarga del tratamiento de datos y del envío de mensajes y está formada por la configuración *fuelgas.nc* y *fuelgasm.nc*. La de más bajo nivel se encarga de la adquisición de datos a través del convertidor analógico-digital y está formada por la configuración *fcm6812.nc* y el módulo *fcm6812m.nc*.

4.3.2 Alto Nivel. Configuración fuelgas.nc y módulo fuelgasm.nc

La configuración *fuelgas.nc* (ver Figura 19 del Anexo II) y el módulo *fuelgasm.nc* son análogos a la configuración y el módulo de alto nivel del programa para el sensor de CO.

En *fuelgas.nc* los componentes que se utilizan son, además del módulo *fuelgasm* y la configuración de la parte de bajo nivel *fcm6812*, los que se ocupan del funcionamiento de los leds, temporizadores y envío de mensajes, *LedsC*, *TimerC* y *GenericCommPromiscuous* respectivamente.

El cableado indica que el componente *Main* simultáneamente inicia, pone en funcionamiento y para el módulo *fuelgasm* (ver Figura 4.30), el componente de temporización y el de comunicaciones mediante interfaces *StdControl*. *fuelgasm* a su vez implementa el funcionamiento de los *leds*, el envío de mensajes, la temporización, el control de la parte de bajo nivel *fcm6812* y la obtención de datos del convertidor analógico digital mediante la interfaz *gas*, alias de *ADC*.

```
module fuelgasm {
  provides {
    interface StdControl;
  }
  uses {
    interface ADC as gas;
    interface SendMsg;
    interface StdControl as fcmControl;
    interface Timer as Timer1;
  }
}
```

Figura 4.30 Interfaces que provee y usa el módulo *fuelgasm*

El módulo *fuelgasm* provee la interfaz *StdControl* y usa las interfaces *ADC*, *Leds*, *Timer*, *SendMsg* y *fcmControl* que es un alias de *StdControl*.

Al principio de la implementación del módulo *fuelgasm.nc* se definen las variables que se utilizan en el módulo (ver Figura 20 del Anexo II). Estas variables se explicarán según aparezcan en las implementaciones de los comandos, eventos o tareas.

A continuación se implementan los comandos de la interfaz *StdControl* que provee el módulo *fuelgasm* , *init*, *start* y *stop* (ver Figura 21 del Anexo II).

En *init* se inician los *leds* y el subprograma de obtención de datos mediante la interfaz *fcmControl* y luego se rellena la cabecera del paquete de mensaje con los datos por defecto.

En el comando *start* se enciende un temporizador repetitivo de cinco segundos y en el comando *stop* se para el temporizador.

Cada vez que el temporizador se dispara se produce el evento *Timer1.fired* (ver Figura 4.31) en el que se comenzará el subprograma de obtención de datos y se ordenará la extracción del dato mediante el comando *getData* de la interfaz *ADC* aquí llamada *gas*.

```
event result_t Timer1.fired()
{
  call fcmControl.start();
  call gas.getData();

  return SUCCESS;
}
```

Figura 4.31 Implementación del evento *fired* de la interfaz *Timer1*

Cuando se haya obtenido el dato se recibirá el evento *dataReady* (ver Figura 4.32) de la interfaz *gas*. En él se trata el dato obtenido y se ordena el envío del mensaje.

```
async event result_t gas.dataReady(uint16_t data) {
  atomic h = (float)data;
  atomic h2 = (((((h/1024)*3)-1)/(3.75))*15000);
  atomic h2ppm = (uint16_t)h2;
  atomic pack->xData.datap1.hidrogeno=h2ppm;
  atomic met = (float)data;
  atomic metan = (((((h/1024)*3)-1)/(3))*15000);
  atomic metanppm = (uint16_t)metan;
  atomic pack->xData.datap1.metano=metanppm;
  atomic prop = (float)data;
  atomic propan = (((((h/1024)*3)-1)/(4))*13250);
  atomic propanppm = (uint16_t)propan;
  atomic pack->xData.datap1.propano=propanppm;
  atomic iso = (float)data;
  atomic isobut = (((((h/1024)*3)-1)/(4))*12000);
  atomic isobutppm = (uint16_t)isobut;
  atomic pack->xData.datap1.isobutano=isobutppm;
  post SendData();
  return SUCCESS;
}
```

Figura 4.32 Implementación del evento *dataReady* de la interfaz *gas*

Lo que se hace es traducir el dato del convertidor analógico-digital que es un número de 0 a 1024 a concentración en partes por millón de los gases que puede leer el sensor ya sea hidrógeno, metano, propano o isobutano. Este número se almacena primero en variables tipo *float*, que son *h*, *met*, *prop* e *iso*.

Luego se divide el dato entre 1024 y se multiplica por 3 (1024 equivale a 3V) obteniendo la tensión en voltios. Para pasar de tensión a concentración se utilizan las relaciones entre estas dos magnitudes que se pueden extraer de las gráficas que hay en la hoja de datos. El dato obtenido se almacena en las variables *h2*, *metan*, *propan* e *isobut*.

Hay que tener cuidado con el tipo de variables que se utilizan puesto que si a una variable entera que no puede valer más de 1024 la dividimos por 1024 y guardamos el valor en una variable entera, aunque luego se multiplique se obtiene cero de resultado lo cual no es correcto. Por esta razón para el tratamiento de los datos es conveniente pasar las variables a tipo *float* y cuando se tenga el resultado final se pasará a entero sin

perder información. Las variables de tipo entero en las que se almacena el resultado final en partes por millón son, *h2ppm*, *metanppm*, *propanppm* e *isobutppm*.

Así se obtienen las cuatro concentraciones y se guardan en el paquete de mensajes. Luego se ordena la tarea *SendData* (ver Figura 4.33) en la que se evalúa si una variable booleana (*sending_packet*) tiene valor verdadero. Si lo tiene es que ya se está mandando un mensaje y se para la tarea ahí, si no asigna valor verdadero a la variable booleana y ordena el envío del mensaje mediante el comando *send* de la interfaz *SendMsg*. Si el comando no se ejecuta correctamente se asigna a la variable booleana el valor falso para indicar que ya no se está enviando ningún mensaje.

```
void task SendData(){
  call fcmControl.stop();
  if (sending_packet) return;
    atomic sending_packet = TRUE;
  if (call SendMsg.send(TOS_BCAST_ADDR,sizeof(XDataMsg),&msg_buffer)!= SUCCESS)
    sending_packet = FALSE;
  return;
}
```

Figura 4.33 Tarea *SendData*

Si el mensaje se ha enviado correctamente se recibe el evento *sendDone* (ver Figura 4.34) de la interfaz *SendMsg*, en él se asignará valor falso a la variable booleana para indicar que ya no estamos enviando el mensaje.

```
/**Evento que avisa que se ha enviado un mensaje correctamente**/
event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {
  atomic sending_packet = FALSE;

  return SUCCESS;
}
```

Figura 4.34 Implementación del evento *SendDone* de la interfaz *SendMsg*

4.3.3 Bajo nivel. Configuración *fcm6812.nc* y módulo *fcm6812m.nc*

La parte de bajo nivel del programa que se dedica a la obtención del dato a través del convertidor analógico-digital está formada por la configuración *fcm6812.nc* (ver Figura 22 del Anexo II) y el módulo *fcm6812m.nc*.

El código de la configuración *fcm6812* es totalmente análogo al código de la configuración *CDM4161* del sensor de CO_2 y lo mismo ocurre con el módulo *CDM4161M* del sensor de CO_2 y el *fcm6812m* (ver Figura 4.35) de éste sensor.

```
module fcm6812m
{
    provides interface StdControl as fcmStdControl;
    provides interface ADC as ExternalfcmADC;
    uses
    {
        interface ADCControl;
        interface ADC as InternalfcmADC;
        interface StdControl as TimerControl;
        interface Timer as fcmTimer;
    }
}
```

Figura 4.35 Interfaces que provee y usa el módulo *fcm6812m*

El módulo *fcm6812m* implementa los comandos de las interfaces que provee, *StdControl* y *ADC*. Así implementa los comandos *init*, *start* y *stop* (ver Figura 23 del Anexo II) de *StdControl*.

En *init* se selecciona el puerto del convertidor que se va a utilizar mediante el comando *bindPort* de la interfaz *ADCControl* y luego se inicia *ADCC* mediante una llamada al comando *init de* la interfaz *ADCControl*. También se inicia componente de temporización *TimerC*. En los comandos *start* y *stop* no se hace nada.

También tienen que implementarse los comandos de *ADC* que son *getData* (ver Figura 4.36) y *getContinuousData*. En getContinuousData no se hace nada puesto que no va a ser llamada nunca. En *getData* se ordena la obtención de una muestra mediante la tarea *getSample*.

```
async command result_t ExternalfcmADC.getData()
{
    post getSample();
    return SUCCESS;
}
```

Figura 4.36 Implementación del comando *getData* de la interfaz *ExternalfcmADC*

La tarea *getSample* (ver Figura 4.37) para el temporizador si este está corriendo y luego comienza otro temporizador de un solo disparo y de 10 tics de reloj y si el temporizador falla vuelve a invocar la tarea *getSample*.

```
// Obtiene la siguiente muestra
task void getSample()
{
    call fcmTimer.stop(); // Sólo si es necesario
    if (call fcmTimer.start(TIMER_ONE_SHOT, 10) != SUCCESS)
    {
        post getSample();
    };
    return;
}
```

Figura 4.37 Tarea *getSample*

Cuando el temporizador iniciado en la tarea termina se produce el evento *fired* (ver Figura 4.38) en el que se ordena al componente *ADCC* la obtención del dato a través del puerto mediante *InternalfcmADC.getData()* y si se produce correctamente la obtención del dato se pone al *Mica2* en espera un pequeño tiempo.

```
// Despues de esperar un poco se toma una lectura
event result_t fcmTimer.fired()
{
    if (call InternalfcmADC.getData() == SUCCESS)
    {
        TOSH_uwait(1000);
        return SUCCESS;
    };
    return SUCCESS;
}
```

Figura 4.38 Implementación del evento *fired* de la interfaz *fcmTimer*

El componente *ADCC* obtendrá entonces el dato del puerto y enviará el evento *InternalfcmADC.dataReady* (ver Figura 4.39) con el dato que se ha extraído que lo que hace es producir la señal *ExternalfcmADC.dataReady* que también contiene el dato.

```
async event result_t InternalfcmADC.dataReady(uint16_t data)
{
    return signal ExternalfcmADC.dataReady(data);
}
```

Figura 4.39 Implementación del evento *dataReady* de la interfaz *InternalfcmADC*

Esta señal produce el evento *ExternalfcmADC.dataReady* (ver Figura 4.40) que en este módulo no hace realmente nada pero que también será recibida por el módulo *fuelgasm* en el evento *gas.dataReady* ya que los componentes están cableados. El evento *gas.dataReady* es el que recibe el dato, lo procesa y ordena el envío.

```
default async event result_t ExternalfcmADC.dataReady(uint16_t data)
{
    return SUCCESS;
}
```

Figura 4.40 Implementación del evento *dataReady* de la interfaz *ExternalfcmADC*

4.3.4 Archivo auxiliar *sensorboardApp.h*

Para completar el programa se tiene, al igual que en los programas de los demás sensores, un archivo auxiliar llamado *sensorboardApp.h* (ver Figura 24 del Anexo II). En este archivo se encuentra la definición del paquete de mensaje y de cadenas de mensajes que en realidad son números que utilizaremos en la asignación de puertos para la conversión analógica-digital.

Al principio se definen las cadenas de caracteres *TOS_ADC_FUELGAS_PORT* y *TOS_ACTUAL_FUELGAS*_PORT como número 3, lo que indica que se utiliza el puerto *ADC3* para la conversión analógica digital.

El paquete de mensajes tiene una cabecera con variables que sirven para identificar el mensaje (en este caso el nodo que envía los datos) y una parte de datos formada por las variables *hidrogeno*, *metano*, *propano* e *isobutano*. Cada una de estas variables contendrá el valor de la concentración en partes por millón del gas que le da nombre.

También se define la cadena de caracteres *AM_XSXMSG* como cero. Esta cadena como se vio en el capítulo 2 es un argumento de la interfaz *SendMsg* y se almacena el campo *type* de los mensajes de *TinyOS* indicando el tipo de mensaje.

4.4 Módulo de control de la calidad del aire: *AM-1*

4.4.1 Descripción general del programa

La aplicación que se ha programado para el módulo para el control de la calidad de aire *AM-1* cuenta con los terminales de salida del microprocesador *A*, *C*, *D* y *E* como entradas de interrupción que se activan con un flanco de bajada. Estos terminales se corresponden respectivamente con el aire limpio, contaminación baja, contaminación media y contaminación alta.

Cuando el aire está en uno de estos niveles el terminal respectivo tiene una tensión de cero voltios (nivel bajo) y el resto de terminales cinco voltios (nivel alto). Por tanto la detección de un flanco de bajada en un terminal (paso de cinco a cero voltios) genera una interrupción y indica al programa en qué nivel se encuentra el aire. A continuación se generará un mensaje con el dato del nuevo nivel de contaminación y se enviará a un *Mica2* receptor. También se enviará un mensaje con el estado cada cinco segundos.

Como en los programas de los sensores anteriores hay dos partes. La de más alto nivel se encarga de la determinación del estado y envío de mensajes y está formada por la configuración *Polut.nc* y el módulo *PolutM.nc*. La parte de bajo nivel se encarga del tratamiento de las interrupciones y está formada por la configuración *PolutSensor.nc* y *PolutSensorM.nc*. Ambas partes se comunican mediante la interfaz *PolutControl.nc*.

4.4.2 Alto nivel. Configuración Polut y módulo PolutM

La configuración *Polut* (ver Figura 25 del Anexo II) de la parte de alto nivel está formada por los componentes *Main*, el módulo *PolutM*, la configuración de bajo nivel *PolutSensor*, el componente de temporización *TimerC* y el de comunicaciones *GenericCommPromiscuous* cuyo alias es *Comm*.

Según el cableado el módulo *PolutM* y el de comunicaciones se inician, se ponen en funcionamiento y se paran simultáneamente mediante la interfaz *StdControl* por el componente principal *Main*.

PolutM controlará el envío de mensajes, la temporización y el funcionamiento de la parte de bajo nivel mediante la interfaz *PolutControl* (ver Figura 26 del Anexo II).

Por tanto el módulo *PolutM* (ver Figura 4.41) provee la interfaz *StdControl* y usa las interfaces *SendMsg*, *PolutControl* y *TimerClock* que es un alias de *Timer*.

```
module PolutM {
  provides {
    interface StdControl;
  }
  uses {
    interface SendMsg;
    interface PolutControl;
    interface Timer as TimerClock;
  }
}
```

Figura 4.41 Interfaces que provee y usa el módulo *PolutM*

Al principio del módulo *PolutM* (ver Figura 4.42) se declaran las variables que se utilizan durante el programa tales como el buffer y el paquete de mensajes, la variable booleana que según tenga valor verdadero o falso indica que se está mandando o no un mensaje y cuatro variables (*Polux0-3*) para indicar el grado de contaminación, una por estado. *Polux0* para el estado de aire limpio, *Polux1* para el de contaminación baja, *Polux2* para el de contaminación media y *Polux3* para el de contaminación alta.

```
implementation {
  bool sending_packet = FALSE;
  TOS_Msg msg_buffer;
  XDataMsg *pack;
  uint8_t Polux0;
  uint8_t Polux1;
  uint8_t Polux2;
  uint8_t Polux3;
```

Figura 4.42 Declaración de las variables que utiliza el módulo *PolutM*

PolutM debe implementar los comandos *init* (ver Figura 27 del Anexo II), *start* y *stop* (ver Figura 4.43) de la interfaz *StdControl* que provee.

En el comando *init* se inicializa el paquete de mensajes rellenando los datos de la cabecera, y se inicializan las variables de estado todas a cero excepto *Polux0* a uno, lo que indica que inicialmente el estado es de aire limpio.

```
/**Llama al comando que habilita las interrupciones**/
command result_t StdControl.start() {
  call PolutControl.start();

  call TimerClock.start(TIMER_REPEAT,5*1024);
  return SUCCESS;
}
/**Llama al comando que deshabilita las interrupciones**/
command result_t StdControl.stop() {
  call PolutControl.stop();
  return SUCCESS;
}
```

Figura 4.43 Implementación de los comandos *start* y *stop* de la interfaz *StdControl*

En el comando *start* se habilitan las interrupciones mediante el comando *PolutControl.start* y se inicia un temporizador repetitivo de cinco segundos.

En el comando *stop* se deshabilitan las interrupciones parando mediante el comando *PolutControl.stop*. Cada vez que el temporizador termina se produce el evento *TimerClock.fired* (ver Figura 4.44).

```
  event result_t TimerClock.fired()
{
  atomic pack->xData.datap1.polut0 = Polux0;
  atomic pack->xData.datap1.polut1 = Polux1;
  atomic pack->xData.datap1.polut2 = Polux2;
  atomic pack->xData.datap1.polut3 = Polux3;
  post SendPolux();
  return SUCCESS;
}
```

Figura 4.44 Implementación del evento *fired* de la interfaz *TimerClock*

En el evento se guarda el valor de cada una de las variables de estado en el paquete de mensajes y luego se ordena la tarea *SendPolux* (ver Figura 4.45).

```
/**Tarea que manda mensajes a través de la radio**/
void task SendPolux(){
  if (sending_packet) return;
    atomic sending_packet = TRUE;
  if (call SendMsg.send(TOS_BCAST_ADDR,sizeof(XDataMsg),&msg_buffer)!= SUCCESS)
    sending_packet = FALSE;
  return;
}
```

Figura 4.45 Tarea *SendPolux*

La tarea evalúa si la variable booleana tiene valor verdadero, si no tiene sigue adelante y le asigna valor verdadero. Entonces ordena el envío del mensaje a través de la radio y si este no se produce correctamente asignamos valor falso a la variable booleana para indicar que ya no se está enviando un mensaje. Si el envío se produce correctamente se recibe el evento *SendMsg.sendDone* (ver Figura 4.46).

```
/**Evento que avisa que se ha enviado un mensaje correctamente**/
event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {
  atomic sending_packet = FALSE;

  return SUCCESS;
}
```

Figura 4.46 Implementación del evento *sendDone* de la interfaz *SendMsg*

En el evento se asigna valor falso a la variable booleana puesto que ya no se está enviando un mensaje.

Cuando cambia el estado del aire se reciben eventos de la interfaz *PolutControl* que cambian el valor de las variables de estado. Esos eventos que se reciben son *PolutControl.fired0* si el estado ha cambiado a aire limpio, *PolutControl.fired1* si ha pasado a contaminación baja, *PolutControl.fired2* si ha pasado a contaminación media y *PolutControl.fired3* si ha pasado a contaminación alta. Por ejemplo si ha pasado a contaminación baja se produce este evento (ver Figura 4.47).

```
event result_t PolutControl.fired1(){
atomic Polux0 = 0;
atomic Polux1 = 1;
atomic Polux2 = 0;
atomic Polux3 = 0;
atomic pack->xData.datap1.polut0 = 0;
atomic pack->xData.datap1.polut1 = Polux1;
atomic pack->xData.datap1.polut2 = 0;
atomic pack->xData.datap1.polut3 = 0;
post SendPolux();
return SUCCESS;
}
```

Figura 4.47 Implementación del evento *fired1* de la interfaz *PolutControl*

Se asigna valor cero a todas las variables de estado, excepto a la que se corresponde con el estado al que ha cambiado, en este caso *Polux1* a la que se le asigna valor uno. Se guardan los datos en el paquete de mensajes y se ordena el envío con la tarea *SendPolux* que ya se explico antes. Por tanto se mandan mensajes cada vez que el sistema cambia de estado y cada cinco segundos. Los eventos para las otras tres interrupciones son análogos a éste.

4.4.3 Bajo nivel. Configuración *PolutSensor* y módulo *PolutSensorM*

En la parte de bajo nivel se dedica al tratamiento de las interrupciones y está formada por la configuración *PolutSensor.nc* (ver Figura 28 del Anexo II) y el módulo *PolutSensorM.nc*.

Solamente provee la interfaz *PolutControl* con la que se comunica con la parte de alto nivel y el cableado *PolutControl=PolutSensorM* indica que es el módulo *PolutSensorM* (ver Figura 4.48) el que provee la interfaz *PolutControl* a todos los efectos.

```
module PolutSensorM {
  provides {
    interface PolutControl;
  }
}
```

Figura 4.48 Interfaz que provee el módulo *PolutSensorM*

PolutSensorM debe implementar los comandos *start* (ver Figura 29 del Anexo II) y *stop* de la interfaz *PolutControl* y el tratamiento de las notificaciones de interrupción.

En el comando *PolutControl.start* se habilitan las interrupciones para los cuatro pines del *INT0* al *INT3* que están conectados a los terminales del sensor. Cada pin tiene un alias: se llama *NON* al *INT3* que está conectado al terminal que indica aire limpio, *LOW* al *INT2* conectado al terminal que indica contaminación baja, *MED* al *INT1* conectado al terminal que indica contaminación media y *HIG* al *INT0* conectado al terminal que indica contaminación alta.

Cada uno de los pines se pone primero a cero mediante la orden *TOSH_CLR_(alias)_PIN()* y se asignan como entrada mediante la orden *TOSH_MAKE_(alias)_INPUT()*.

Luego se selecciona interrupción por flanco de bajada mediante la asignación de valores al registro de control de interrupciones externas *B* (*EICRB*). Para cada pin se pondrá a uno *ISCn1* y se pondrá a cero *ISCn0* siendo *n* un número que se corresponde con cada pin de esta manera, para *INT0* 4, para *INT1* 5, para *INT2* 6 y para *INT3* 7. Finalmente se habilitan las interrupciones poniendo a uno la máscara de interrupciones externas (*EIMSK*) mediante la orden *sbi(EIMSK,n)*.

En el comando *PolutControl.stop* (ver Figura 4.49) se deshabilitan todas las interrupciones poniendo a cero el registro *EIMSK* para cada pin de interrupción.

```
/**Deshabilita las interrupciones**/
command result_t PolutControl.stop() {
  cbi(EIMSK,4);                    // Deshabilita las interrupciones en INT0
  cbi(EIMSK,5);                    // Deshabilita las interrupciones en INT1
  cbi(EIMSK,6);                    // Deshabilita las interrupciones en INT2
  cbi(EIMSK,7);                    // Deshabilita las interrupciones en INT3
  return SUCCESS;
}
```

Figura 4.49 Implementación del comando *stop* de la interfaz *PolutControl*

Cuando se produce un flanco de bajada en alguno de los pines le llega a *PolutSensorM* una señal de interrupción que es tratada en *TOSH_SIGNAL(SIG_INTERRUPTn)* (ver Figura 4.50) siendo *n* igual que antes el número que identifica el pin.

```
TOSH_SIGNAL(SIG_INTERRUPT5)
{
  cbi(EIMSK,5);                    // Deshabilita interrupciones
  signal PolutControl.fired2();
  sbi(EIMSK,5);                    // Habilita interrupciones
}
```

Figura 4.50 Tratamiento de la señal de interrupción de *INT1*

En el tratamiento de la señal se pone a cero la máscara de interrupciones externas deshabilitando así las interrupciones y se manda la señal a la parte de alto nivel de que en ese pin se ha producido una interrupción, seguidamente se vuelven a habilitar las interrupciones en ese pin poniendo a uno la máscara de interrupciones externas.

4.4.4 Archivo auxiliar *sensorboardApp.h*

Para completar el programa se necesita el archivo auxiliar *sensorboardApp.h* (ver Figura 30 del Anexo II) en el que se encuentra la definición del paquete de mensajes y la asignación de alias a los pines.

La asignación de nombre a los pines se hace mediante el comando *TOSH_ALIAS_PIN* y permite cambiar de pin en el archivo auxiliar sin tener que tocar el programa. Como ya se ha dicho al pin de interrupción *INT3* le corresponde el alias *NON*, al *INT2 LOW*, al *INT1 MED* y al *INT0 HIG*.

El paquete de mensajes está dividido como siempre en cabecera y datos. En la cabecera se encuentran las variables que definen la procedencia del mensaje, aunque en este caso

la única variable que puede cambiar de un mensaje a otro es *node_id* que indica cual de los nodos *Mica2* ha enviado el mensaje. La parte de datos contendrá el valor de las variables que se han denominado de estado. De todas sólo una contiene valor uno y las demás cero, la que está con valor uno nos indica en qué estado está el aire.

4.5 Aplicación conjunta de los cuatro sensores

4.5.1 Descripción general del programa

Por último se ha desarrollado una aplicación que comprende los cuatro sensores que ya se han visto: El sensor de monóxido de carbono *TGS5042*, el sensor de dióxido de carbono *CDM4161*, el Sensor de hidrógeno, metano y gases licuados del petróleo *FCM6812* y el módulo de control de la calidad del aire *AM-1*.

La aplicación leerá desde distintas entradas analógicas-digitales el valor de los sensores de *CO*, CO_2 y H_2 y por otro lado los terminales del módulo *AM-1* se conectan a las cuatro entradas de interrupción de la misma manera que en la aplicación del apartado 4.4. Así pues, según cuál de las cuatro entradas de interrupción esté a nivel bajo se sabrá cual es su nivel de contaminación.

Cada cinco segundos se accederá al contenido de los convertidores analógico-digitales traduciendo el dato obtenido a sus respectivas unidades y se determinará el estado de contaminación. Seguidamente se enviará un mensaje conteniendo los datos adquiridos.

Como en los programas de los sensores anteriores hay dos partes. La de más alto nivel se encarga de pasar los datos obtenidos a través de los convertidores analógicos-digitales, de la determinación del estado de contaminación y del envío de mensajes y está formada por la configuración *Medidor.nc* y el módulo *MedidorM.nc*. La parte de bajo nivel está formada por el conjunto de las partes de bajo nivel de las cuatro aplicaciones anteriores.

Por parte de la aplicación del sensor de *CO* se encuentran la configuración *COSensor* y el módulo *COSensorM*, por parte de la aplicación del sensor de CO_2 se encuentran la configuración *CDM4161* y el módulo *CDM4161M*, por parte de la aplicación del sensor de H_2 se encuentran la configuración *fcm6812* y el módulo *fcm6812m*. Estos archivos se encargan de la adquisición de datos de los convertidores analógicos digitales. Por parte de la aplicación del módulo de control de calidad del aire está la configuración *PolutSensor.nc* y el módulo *PolutSensorM.nc*. Estas partes se encargan del tratamiento de las interrupciones y se comunican con la parte de alto nivel mediante la interfaz *PolutControl.nc*.

4.5.2 Alto nivel. Configuración *Medidor* y módulo *MedidorM*

La configuración *Medidor* (ver Figura 31 del Anexo II) de la parte de alto nivel está formada por los componentes *Main*, el módulo *MedidorM*, las configuraciones de bajo nivel *PolutSensor*, *COSensor*, *CDM4161* y *fcm6812*, el componente de temporización *TimerC* y el de comunicaciones *GenericCommPromiscuous* cuyo alias es *Comm*.
Según el cableado el módulo *MedidorM*, el de temporización y el de comunicaciones se inician, se ponen en funcionamiento y se paran simultáneamente mediante la interfaz *StdControl* por el componente principal *Main*.
MedidorM controlará el envío de mensajes, la temporización y el funcionamiento de las partes de bajo nivel.
Por tanto el módulo *MedidorM* (ver Figura 4.51) provee la interfaz *StdControl* y usa las interfaces *SendMsg*, la interfaz *ADC* con los alias *CODOS*, *Gas* y *gas*, la interfaz *StdControl* con los alias *CDMControl*, *COControl* y *fcmControl*, *PolutControl* y *Timer*.

```
module MedidorM {
  provides {
    interface StdControl;
  }
  uses {
    interface ADC as CODOS;
    interface ADC as Gas;
    interface ADC as gas;
    interface PolutControl;
    interface StdControl as CDMControl;
    interface StdControl as COControl;
    interface StdControl as fcmControl;
    interface Timer;
    interface SendMsg as Send;
  }
}
```

Figura 4.51 Interfaces que provee y usa el módulo *MedidorM*

Al principio del módulo *MedidorM* (ver Figura 4.52) se declaran las variables que se utilizan durante el programa tales como el buffer y el paquete de mensajes, la variable booleana que según tenga valor verdadero o falso indica que se está mandando o no un mensaje, la variable *Polux* que según su valor indica el grado de contaminación, *0* para el estado de aire limpio, *1* para el de contaminación baja, *2* para el de contaminación media y *3* para el de contaminación alta, las variables que se utilizan para pasar los datos de los convertidores analógicos-digitales a concentración en ppm de los distintos gases y las variables *ready1*, *ready2* y *ready3* que indican si un dato de un convertidor analógico-digital ha sido ya extraído.

```
implementation {
//Lista de variables que se usan en el módulo
  bool sending_packet = FALSE;
  TOS_Msg msg_buffer;
  XDataMsg *pack;
  float adc;
  float volts;
  float ppm;
  uint16_t PPM;
  float co2;
  float co22;
  uint16_t co2ppm;
  float h;
  float h2;
  uint16_t h2ppm;
  uint8_t Polux;
  uint8_t ready1;
  uint8_t ready2;
  uint8_t ready3;
}
```

Figura 4.52 Declaración de las variables que utiliza el módulo *MedidorM*

MedidorM debe implementar los comandos *init* (ver Figura 32 del Anexo II), *start* y *stop* (ver Figura 4.53) de la interfaz *StdControl* que provee.

En el comando *init* se inicializa el paquete de mensajes rellenando los datos de la cabecera, y se inicializan la variable de estado *Polux* y las *ready1-3* a cero lo que indica que inicialmente el estado es de aire limpio y todavía no se han extraído los datos de los convertidores analógicos-digitales. También se inician las partes de bajo nivel de los sensores de *CO*, CO_2 y H_2 mediante las interfaces *COControl*, *fcmControl* y *CDMControl*.

```
  /**
   * Comienza el componente y habilita las interrupciones
   **/
command result_t StdControl.start(){
    call PolutControl.start();
    // Comienza un temporizador repetitivo que se dispara cada 5 segundos
    return call Timer.start(TIMER_REPEAT, 5*1024);
}

  /**
   * Para la ejecución de la aplicación y deshabilita las interrupciones.
   * Para también el componente de temporización.
   **/
command result_t StdControl.stop(){
    call PolutControl.stop();
    return call Timer.stop();
}
```

Figura 4.53 Implementación de los comandos *start* y *stop* de la interfaz *StdControl*

En el comando *start* se habilitan las interrupciones mediante el comando *PolutControl.start* y se inicia un temporizador repetitivo de cinco segundos.

En el comando *stop* se deshabilitan las interrupciones parando mediante el comando *PolutControl.stop* y se para el temporizador. Cada vez que el temporizador termina se produce el evento *Timer.fired* (ver Figura 4.54).

```
event result_t Timer.fired(){
    call COControl.start();
    call CDMControl.start();
    call fcmControl.start();
    call Gas.getData();
    call CODOS.getData();
    call gas.getData();
}
```

Figura 4.54 Implementación del evento *fired* de la interfaz *Timer*

En el evento se comienzan las partes de bajo nivel de los sensores de *CO*, CO_2 y H_2 mediante las interfaces *COControl*, *fcmControl* y *CDMControl* y se ordena la obtención de los datos mediante los alias de la interfaz *ADC*.

En respuesta a las llamadas a los alias de la interfaz *ADC* y cuando los datos de los distintos sensores estén listos, se recibirán los eventos *dataReady* (ver Figuras 33, 34 y 35 del Anexo II). En ellos se procesará el dato obtenido hasta obtener el valor de concentración en partes por millón del respectivo gas y se almacenará el resultado en el paquete de mensaje. Además se pondrá a 1 su respectiva variable *ready*, *ready1* para el sensor de *CO*, *ready2* para el sensor de H_2 y *ready3* para el sensor de CO_2. Al término de cada evento se llama a la ejecución de la tarea *Ready* (ver Figura 4.55).

```
    /**
     * Tarea que comprueba si los datos están listos
     * y en ese caso ordena la tarea SendData que envía
     * el mensaje.
     **/
void task Ready(){
    if (ready1==1 && ready2==1 && ready3==1){
        atomic ready1=0;
        atomic ready2=0;
        atomic ready3=0;
        atomic pack->xData.datap1.polux = Polux;
        post SendData();
    }
    return;
}
```

Figura 4.55 Tarea *Ready*

La tarea evalúa si todas las tres variables *ready* valen uno, es decir, si ya están disponibles los tres datos obtenidos de los convertidores analógicos-digitales. Si es así se ponen a cero las tres variables *ready*, se guarda el valor de la variable *Polux*, que indica el nivel de contaminación, en el paquete de mensaje y se ordena la tarea de envío de mensajes *SendData* (ver Figura 4.56).

```
    /**
     * Tarea que para los componentes COControl, CDMControl y fcmControl
     * y envía el mensaje.
     **/
void task SendData(){
    call COControl.stop();
    call CDMControl.stop();
    call fcmControl.stop();

    if (sending_packet) return;
    atomic sending_packet = TRUE;

    // Lanza un mensaje vía radio
    if(call Send.send(TOS_BCAST_ADDR,sizeof(XDataMsg),&msg_buffer)!=SUCCESS)
      sending_packet = FALSE;

    return;
}
```

Figura 4.56 Tarea *SendData*

La tarea para las partes de bajo nivel de los sensores de *CO*, CO_2 y H_2 y evalúa si la variable booleana tiene valor verdadero, si no tiene sigue adelante y le asigna valor verdadero. Entonces ordena el envío del mensaje a través de la radio y si este no se produce correctamente asignamos valor falso a la variable booleana para indicar que ya no se está enviando un mensaje. Si el envío se produce correctamente se recibe el evento *Send.sendDone* (ver Figura 4.57).

```
    /**
     * Evento que indica que el mensaje periódico ha sido enviado correctamente
     **/
event result_t Send.sendDone(TOS_MsgPtr msg, result_t success) {
  atomic sending_packet = FALSE;
  return SUCCESS;
}
```

Figura 4.57 Implementación del evento *sendDone* de la interfaz *Send*

En el evento se asigna valor falso a la variable booleana puesto que ya no se está enviando un mensaje.

Cuando cambia el estado del aire se reciben eventos de la interfaz *PolutControl* que cambian el valor de las variables de estado. Esos eventos que se reciben son *PolutControl.fired0* si el estado ha cambiado a aire limpio, *PolutControl.fired1* si ha pasado a contaminación baja, *PolutControl.fired2* si ha pasado a contaminación media y *PolutControl.fired3* si ha pasado a contaminación alta. Por ejemplo si ha pasado a contaminación baja se produce este evento (ver Figura 4.58).

```
    //Cambia a contaminación baja, asigna un uno
event result_t PolutControl.fired1(){
  atomic Polux=1;
}
```

Figura 4.58 Implementación del evento *fired1* de la interfaz *PolutControl*

Se asigna valor a la variable *Polux* dependiendo de a qué estado se ha pasado, se asigna valor cero si pasa a estado no contaminado, valor uno si pasa a contaminación baja, valor dos si pasa a contaminación media y valor tres si pasa a contaminación alta.

4.5.3 Bajo nivel

Como ya se ha dicho, la parte de bajo nivel está formada por el conjunto de las partes de bajo nivel de las cuatro aplicaciones anteriores. En concreto, por parte de la aplicación del sensor de *CO* se encuentran la configuración *COSensor* y el módulo *COSensorM*, por parte de la aplicación del sensor de CO_2 se encuentran la configuración *CDM4161* y el módulo *CDM4161M*, por parte de la aplicación del sensor de H_2 se encuentran la configuración *fcm6812* y el módulo *fcm6812m* y por parte de la aplicación del módulo de control de calidad del aire está la configuración *PolutSensor.nc* y el módulo *PolutSensorM.nc*. A estos dos últimos archivos hay que sumarle la interfaz *PolutControl.nc* que los comunica con la parte de alto nivel. Todos estos archivos se encuentran descritos en los apartados anteriores de este mismo capítulo.

4.5.4 Archivo auxiliar *sensorboardApp.h*

Para completar el programa se necesita el archivo auxiliar *sensorboardApp.h* (ver Figura 36 del Anexo II) en el que se encuentra la definición del paquete de mensajes ,la asignación de alias a los pines de interrupción y la declaración de cadenas de mensajes que en realidad son números que utilizaremos en la asignación de puertos para la conversión analógica-digital.

La asignación de nombre a los pines de interrupción se hace mediante el comando *TOSH_ALIAS_PIN* y permite cambiar de pin en el archivo auxiliar sin tener que tocar el programa. Como ya se ha dicho al pin de interrupción *INT3* le corresponde el alias *NON*, al *INT2 LOW*, al *INT1 MED* y al *INT0 HIG*.

También se definen las cadenas de caracteres *TOS_ADC_CO_PORT* y *TOS_ACTUAL_CO*_PORT como número 3, *TOS_ADC_CO2_PORT* y *TOS_ACTUAL_CO2*_PORT como número 4 y *TOS_ADC_FUELGAS_PORT* y *TOS_ACTUAL_FUELGAS*_PORT como número 5 lo que indica que se utiliza el puerto *ADC3* para la conversión analógica digital del sensor de *CO*, el puerto *ADC4* para la conversión analógica digital del sensor de CO_2 *y* el puerto *ADC5* para la conversión analógica digital del sensor de H_2.

El paquete de mensajes está dividido como siempre en cabecera y datos. En la cabecera se encuentran las variables que definen la procedencia del mensaje, aunque en este caso la única variable que puede cambiar de un mensaje a otro es *node_id* que indica cual de los nodos *Mica2* ha enviado el mensaje. La parte de datos contendrá el valor de los datos extraídos de los sensores de *CO*, CO_2 y H_2 en las variables *co*, *co2* e *hidro* respectivamente y el estado de contaminación en la variable *polux*.

También se define la cadena de caracteres *AM_XSXMSG* como cero. Esta cadena como se vio en el capítulo 2 es un argumento de la interfaz *SendMsg* y se almacena el campo *type* de los mensajes de *TinyOS* indicando el tipo de mensaje.