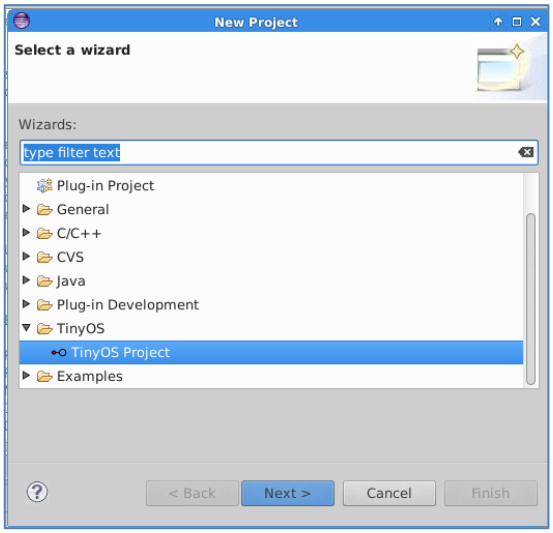
TP TinyOS

On travaillera sur la Machine Virtuelle UBUNTU[1]. Lancer Ubuntu.

1) Allumer une LED

Nous utiliserons l'environnement Eclipse pour programmer en NesC. On va faire clignoter une des leds de la carte Led 0. Dans un premier temps, nous allons juste l'allumer.



Lancer l'application "Eclipse". Ensuite,

File→ new → project …

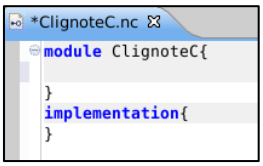
Comme présenté ci-contre.

Sélectionner TinyOS Project et puis "next". Pour le nom du projet, taper "Clignote", et comme target, choisissez "telosb" et ensuite "finish". Cliquez sur "yes" dans la fenêtre qui s'ouvre éventuellement.

Vous avez dans la fenêtre "Package Explorer" le projet "Clignote" qui se crée avec une sous répertoire "src".

Nous allons créer une application pour faire clignoter la LED qui se trouve sur la carte TelosB.

Créer un nouveau "module" : File-> New-> Module. On l'appelle "ClignoteC". Par convention on ajoute un "C" à la fin du nom du projet pour appeler les modules.



Dans toutes les applications écrites pour TinyOs, il existe un component "MainC" qui doit être exécuté en premier. C'est un ensemble des routines nécessaires pour initialiser et configurer différentes parties de la carte, il prend aussi la main et gère les séquencements. Une interface qui nous permet de se rendre compte de la fin de ces configurations s'appelle "boot". On se sert de cette interface-là dans notre module. Pour pouvoir créer cette interface au niveau de module on utilise le mot clé "uses" (vous pouvez utiliser "cntrl + espace" pour voir la liste des interfaces que l'outil vous propose).

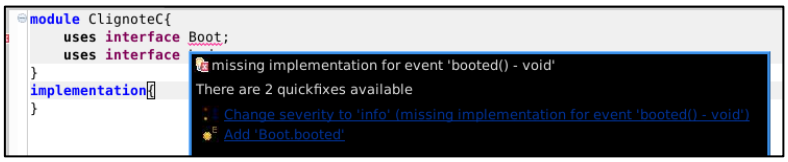
```
module ClignoteC{
    uses interface Boot;
}
```

[1] Login : tp, mot de passe : elec

Les fonctions de l'interface Boot sont implémenté dans un component qui s'appelle « MainC ». On reliera notre interface Boot à celle de MainC.boot dans le fichier de "configuration" plus tard.

Pour pouvoir allumer les LEDs qui se trouvent sur la carte, on utilise le component LedsC qui implémente les fonctions correspondantes. L'interface "Leds" associée propose un ensemble de fonctions que l'on peut appeler. Pour cela on ajoute la ligne : uses interface Leds;

On remarque que "Boot" est souligné en rouge. En mettant le curseur sur cette ligne, et en cliquant sur la ligne "Add Boot.booted" qui s'affiche dans une fenêtre contextuelle, l'outil crée une fonction qui sera exécutée quand l'évènement Boot sera achevé.



Ainsi, l'outil nous génère "event handler" suivant:

```
event void Boot.booted(){
    // TODO Auto-generated method stub
}
```

Une fois « Boot » terminé, nous souhaitons allumer une LED de la carte. Pour mettre en évidence l'utilisation des sous-programme, on procède comme ci-dessous.

On crée un sous-programme pour allumer la LED 0. On appellera ce sous-programme que l'évènement « Boot » a été accompli : c'est à ce moment-là que la fonction Boot.booted() est exécutée, et c'est vous qui implanterez cette fonction :

```
implementation{
	void AllumerLed(){
		call Leds.led0On();
	}
	event void Boot.booted(){
		AllumerLed();
	}
}
```

Bien sûr, il était possible de ne pas passer par un sous-programme, comme :

```
implementation{
	event void Boot.booted(){
		call Leds.led0On();
	}
}
```

Avec ce programme, dès le lancement, la Led0 s'allume. Par contre, le programme n'est pas complet car il faut dire à compilateur quel component utilise quelle interface et dans quel sens. Alors, un fichier de configuration est nécessaire qui introduit ces différents components (Boot, Led, ClignoteC) et présente leurs connections.

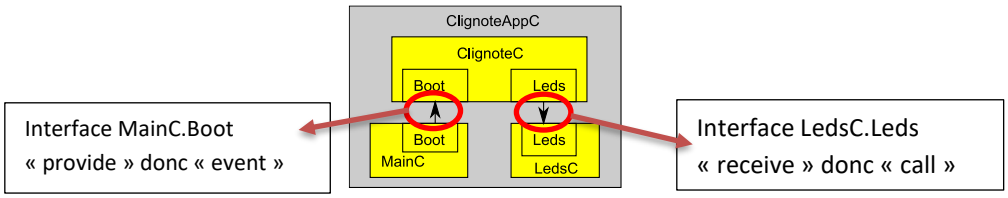
Créer un fichier de configuration : File→ New → configuration et comme nom, donner "ClignoteAppC". On essaye de respecter cette convention Nom du projet + AppC pour désigner les configurations.

On s'intéresse à la partie "implementation". Dans un premier temps on introduit les components qui implémentent les fonctions de nos interfaces. On doit obligatoirement définir le component MainC.

Ensuite, nous déclarons le component LedC qu'on utilisera pour manipuler les LEDs, et le component que nous venons de créer, ClignoteC, qu'on appellera App. Ci-dessous le code à écrire:

```
configuration ClignoteAppC{
}
implementation{
	components MainC;
	components LedsC;
	components ClignoteC as App;
}
```

Maintenant on fait le « câblage » qui indique où nos interfaces sont implémentées. Ci-dessous le schéma de câblage:



Pour faire effectuer ce câblage, on écrit les lignes suivantes:

```
App.Boot -> MainC.Boot;
App.Leds -> LedsC.Leds;
```

La dernière chose qui reste est de créer le "Makefile" pour dire à compilateur comment compiler les fichiers:

File→ New → File et ensuite, saisir "Makefile" comme le nom du fichier.

Dans le fichier Makefile taper ces deux lignes:

```
COMPONENT = ClignoteAppC
include $(MAKERULES)
```

Nous allons maintenant tester le programme. Connectez le "mote" telosB à un port USB du PC. Pour vérifier si le mote a été détecté par le PC, dans une fenêtre de commande taper la commande "motelist". Vous devrez voir affiché /dev/ttyUSB0 qui montre que le mote a été reconnu. Vous devrez aussi autoriser l'accès à ce port en tapant: sudo chmod 666 /dev/ttyUSB0.

Utilisant la commande "cd" aller au répertoire "workspace/Clignote/src" et ensuite vérifier son contenu avec la commande "ls".

Ensuite, saisir la commande "make telosb" qui compile le programme pour la carte telosB et qui ajoute un répertoire appelé "build" dans lequel il existe un répertoire appelé "telosb". Dans ce répertoire vous trouverez les fichiers exécutables.

Aller dans le répertoire "src" et en utilisant la commande "make " pour nettoyer ce répertoire. Cette fois-ci on charge le programme sur la carte telosB. On saisit la commande suivante:

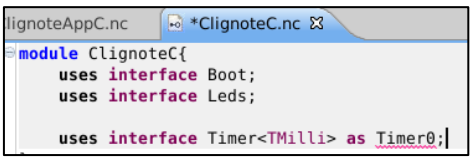
```
>make telosb install
```

Normalement, le code est téléchargé sur la carte et la LED correspondante doit être allumée.

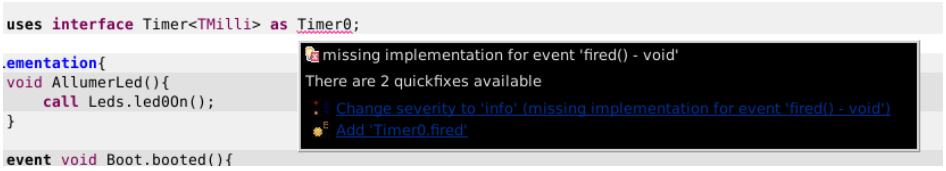
2) Faire clignoter la LED0

Dans un premier temps, on crée un « timer » pour créer un évènement périodique que nous utiliserons pour basculer l'état de la LED: on-off-on-off…

Pour cela, nous ajoutons le component TimerMilliC avec son interface "timer". Aller au fichier "ClignoteC.nc" et ajouter l'interface timer :



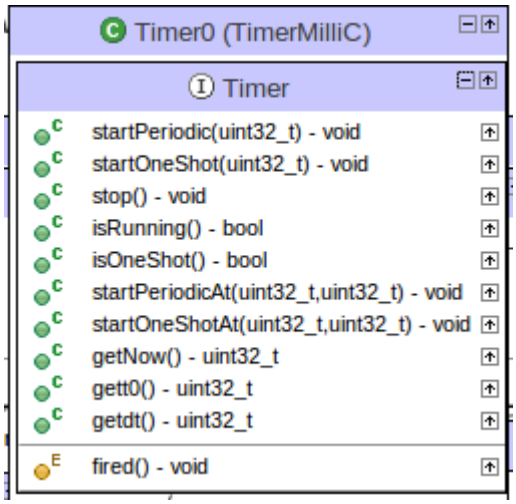
Notez que nous avons instancié un exemple de Timer que l'on appelle "Timer0". Ceci est parce que le timer est un component générique, on peut donc en avoir plusieurs. On a aussi ajouté <TMilli> pour avoir un timer avec une résolution de milliseconde. Remarque: vous pouvez faire afficher la liste des options possibles en utilisant les touches « Ctrl+espace ».



On remarque l'erreur sur "Timer0". Mettre le pointeur de souris sur l'erreur et puis cliquez sur "Add Timer0.fired". Un évènement (event handler) se crée dans la partie implémentation du module. Cet évènement sera exécuté quand le timer arrive à la fin de son comptage. On pourra ajouter une fonction qui fera clignoter la LED ici.

```
event void Timer0.fired(){
    // TODO Auto-generated method stub
}
```

Mais avant, il faudra initialiser le timer pour qu'il déclenche les évènements réguliers. Le meilleur endroit pour initialiser le timer est dès le départ, c'est-à-dire en "boot". On crée un événement tous les 0.5 seconde :




```
event void Boot.booted(){
        call Timer0.startPeriodic(500);
        AllumerLed();
}
```

Maintenant on va basculer la LED à chaque évènement de fin de timer:

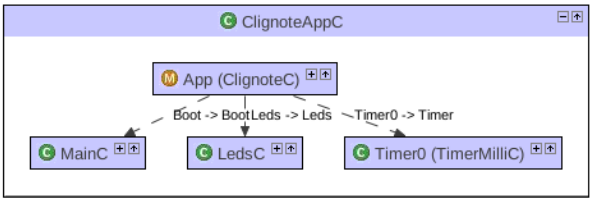
```
event void Timer0.fired(){
        call Leds.led0Toggle();
}
```

Maintenant il faudra annoncer la nouvelle configuration dans le fichier "ClignoteAppC".

```
implementation{
        components MainC;
        components LedsC;
        components new TimerMilliC() as Timer0;
```

```
        components ClignoteC as App;
        // cablage
        App.Boot -> MainC.Boot;
        App.Leds -> LedsC.Leds;
        App.Timer0 -> Timer0;
}
```

Le mot clé "new" est utilisé parce que le timer est un component singleton, on effectue donc une instanciation de ce component en appelant le constructeur. Le graphe de câblage peut être vérifié en choisissant l'onglet "component graph":



Maintenant tout est complet, on va compiler de la même manière qu'avant:

```
>make clean
>make telosb install
```

Exercice: ajouter un autre timer que vous appellerez "timer1". Sur chaque évènement de ce timer, basculer la LED1. Fixer la période de ce timer à 250 ms.

3) Prise en main des buttons

In this TP we would like to create events from the buttons. For that, we use the component UserButtonC related to the buttons. We will turn on and off a LED of the board depending on the button state. You may take a look at the web site[2] in order to see all the available components for telosB. For example, you can check the TimerMilliC component.

Create a new project that you name "UserButtonTest". Add a new module to it with that you will name "UserButtonTestC".

According to the website, the component "UserButtonC" provides the two following interfaces: "Get" and "Notify":

```
interface Get<button_state_t>
interface Notify<button_state_t>
```

Note the data type of the argument of the interface. In order to be able to use the required types, we need to include the file "UserButton.h" at the beginning of the module. Your module should look like:

```
#include<UserButton.h>

module UserButtonTestC{
        uses interface Boot;
        uses interface Leds;
        uses interface Get<button_state_t>;
        uses interface Notify<button_state_t>;
```

[2] https://github.com/tinyos/tinyos-main/tree/master/tos/system

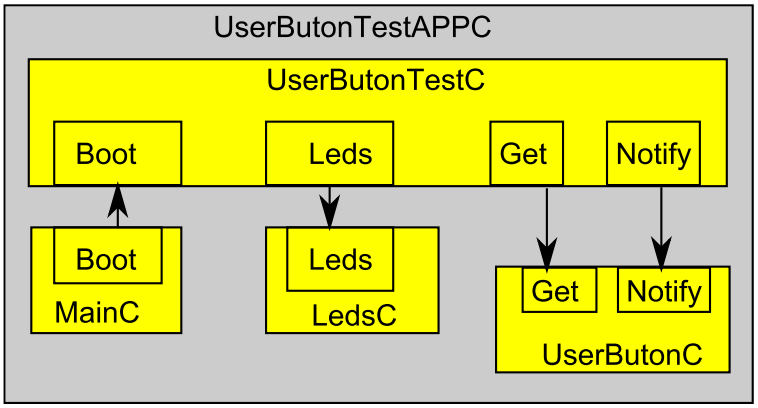
```
}
implementation{
	event void Boot.booted(){
		// TODO Auto-generated method stub
	}
		event void Notify.notify(button_state_t val){
		// TODO Auto-generated method stub
	}
}
```

Now add a configuration file that you will call "UserButtonTestAppC". In this new file, add the component "MainC" that contains the boot interface, add also the main component "UserButtonTestC" (we will call it as App as a short hand).

```
configuration UserButtonTestAppC{
}
implementation{
	components UserButtonTestC as App; //our main module
	components MainC;
}
```

We will use also the component "LedsC", so add also this component and UserButtonC component.

Now we are going to wiring the components through their interfaces. Referring the following figure, the connection will be done in the implementation part:




```
App.Boot -> MainC.Boot;
App.Leds -> LedsC.Leds;
App.Get -> UserButtonC.Get;
App.Notify -> UserButtonC.Notify;
```

Now that the connections are done, we will think about the programming. At the beginning, we should enable the Notify function. For that, in the boot event, we call the Notify.enable function. You can type Cntrl+space to view all the available actions and select the enable function.

Now, we wait for the event, which is a change on the state of the button. Two cases: it is pressed or it is released. The function Notify.notify returns 1 if it is pressed and 0 if it is released. Turn on or off the Led0 depending these two cases.

Now create the Makefile as before:

```
COMPONENT=UserButtonTestAppC
include $(MAKERULES)
```

Compile and download the code and then test it.

4) Writing on PC

To be able to write on PC, the TelosB board should send the ASCII code of the characters via the serial port to the PC. Several things should be done. At the first, you must use the components SerialPrintfC, so add it to your configuration file. Second, you must include to your module the following two header files : stdio.h, string.h. Third, you should add the following line in your Makefile :

```
PFLAGS += -I$(TOSDIR)/lib/printf
```

Now you can send formatted text to the serial port using the command printf. For example

```
printf("%d\r\n", val) ;
```

Will send the numeric value of the variable “val” to the serial port.

At the PC side, in order to monitor the serial port, you should run a serial port interface. You may run the terminal by

Applications -> Accessories -> Terminal

To test your printing capability, write the state of the LED0 on the PC screen:

« The LED is on » or « The LED is off. »

5) Reading a sonsor

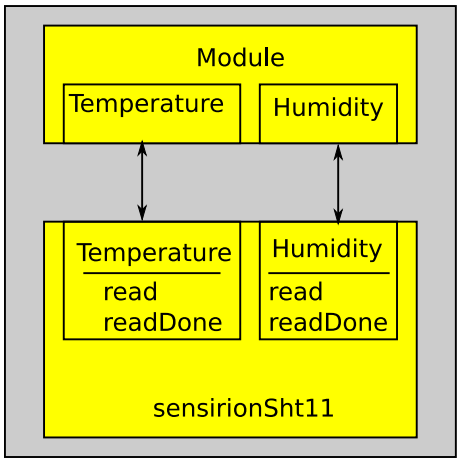
Create a new project that you call SenseTemp. We will read the temperature and write it on PC. On the telosB board, there is a component that manages the temperature and humidity sensors that must be declared in the configuration file:

```
components new SensirionSht11C() as SensorC ;
```

Using this line, the component is renamed as SensorC, where both of the temperature and humidity sensors can be read. Now we use the “Read” interface to read the temperature and the humidity. This can be done by defining two interfaces of “Read”, therefore, in order to dissociate them we need to name them differently as below:

```
uses interface Read<uint16_t> as Temperature;
uses interface Read<uint16_t> as Humidity;
```

To read the temperature we **`call Temperature.read()`** and to read the humidity we use **`call Humidity.read()`**. Once the interfaces are declared in the module, we go to the configuration to do the connection as shown in the figure bellow:



It means:

```
App.Temperature -> SensorC.Temperature;
App.Humidity -> SensorC.Humidity;
```

The Read interface has two functions and work splitted: the read function and the readDone function. Therefore you can, for example use the following line in the Timer.fired() event :

```
Call Temperature.read() ;
Call Humidity.read();
```

Then wait for the Temperature.readDone and Humidity.readDone events as below:

```
event void Temperature.readDone(error_t result, uint16_t data)
  {
    if (result == SUCCESS){ printf("T=%d\r\n",val);}
  }
```

Do the same for the Humidity.readDone. The values printed should be converted to the true values in % and in °C. To compute the temperature value from the raw data read by the sensor we use the formula:

$$T = 0.01 \times read_{value} - 39.6\,°C$$

Note: TinyOS cannot print the floating data. You can use the following function to print floating data:

```
Declare at the beginning of the implementation:
double  temp;
uint8_t integ, frac;

Then add the function:
void printFloat(double data)
{
       if (data<0){ data=-data;  printf("-"); }
       integ = data;
       frac=(data-integ)*100+0.5;
       printf("%d.%d°C",integ,frac);
}
```

Now to print the temperature in degree at the Temperature.readDone:

```
temp = (double) val * 0.01 – 39.6;
printf("T=");printFloat(temp);print("°C\r\n");
```

For the humidity the formula is

$$H = -2.0468 + 0.0367val - 1.5955 \times 10^{-6} val^2$$

A simple program could be:

```
temp = -2.0468 + val *(0.0367 - 1.5966e-6 * val);
```

6) Sending a packet using ZigBee standard

In this exercise, we would like to send a message to all the motes at the range. First, we must prepare the packet type and the information that it contains. To do that, we will create a header file in which we define a structure with all the required fields.

Create a new project and call it MoteToMote.

Then create a .h file where we will put the required type:

File → new →header file and then name it MoteToMote.

We will define a structure containing the data that will be transmitted between the motes. Since each mote should have an identifier (address), we create an 8-bit field that we name as "ID", it will be sent along the packet (it is to note that the ID of the source is inserted automatically in the packet, but here we explicitly insert it in the data circulating in the network).

Type the following lines in the header file MoteToMote.h.

```
#ifndef MOTE_TO_MOTE_H
#define MOTE_TO_MOTE_H

typedef nx_struct MyMsg_t{
	nx_uint8_t NodeId;
	nx_uint8_t Temp_int;
	nx_uint8_t Temp_frac ;
	nx_uint8_t Humid;
}MyMsg_t;

#endif /* MOTE_TO_MOTE2_H */
```

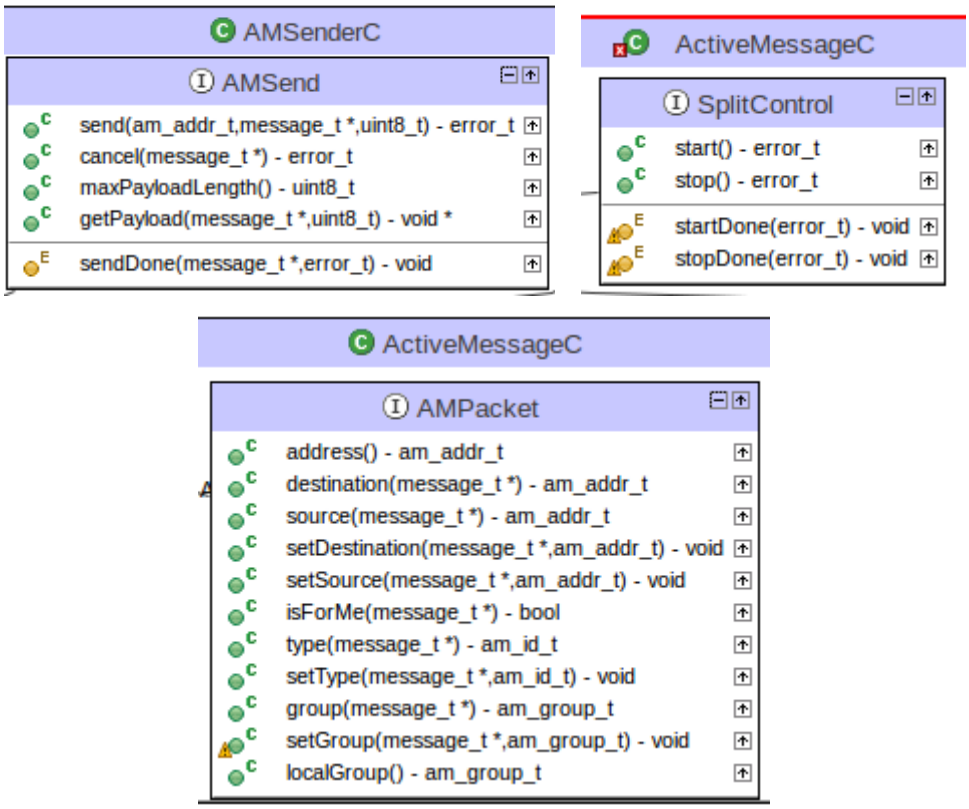
You should include this file to your module using #include directive.

We imagine the following sequencing:

1- In the boot, we turn the radio on
2- In the event "radio-on done", we program a timer with period 8 sec
3- In timer "fired event", we call the read temperature function
4- In "temperature read done" we call the humidity read function

5- In the “humidity read done” we send the packet.

Therefore, we need to use some interfaces that contain functions for timer, for reading the sensors, and sending packets. The interface related to preparing the packet and to sending packets are implemented by two components: ActiveMessageC and AMSenderC, presented below with their corresponding interfaces and functions within. We will use ActiveMessageC.splitControl, ActiveMessageC.AMPacket and AMSenderC.AMSend.



In the above figures, you can discover the interface AMSend, SplitControl and AMPacket. All the functions that you can call and the event that you may handle are given.

Steps to follow:

Create a new module called “MoteToMoteTestC” and a new configuration “MoteToMoteTestAppC”.

Radio preparation: In the module, you should use the Boot interface, as before, and the SplitControl interface (that we rename as AMControl, see below). At the “booted done” event, we use the start() function of SplitControl interface to turn on the radio in split mode.

```
call AMControl.start();
```

Then, we wait for the startDone() event to begin a timer. To be specific we enter the following line in the module:

```
uses interface SplitControl as AMControl;
```

And we do the connection in the configuration file as:

```
App.AMControl -> ActiveMessageC.splitControl;
```

Then, we wait the event “AMControl.startDone”. In event handler

```
event void AMControl.startDone(error_t error)
```

we check the variable “error”, if it is SUCCESS it means that everything is OK and we start a periodic timer to send a message periodically (say every 8 seconds) as seen in section (2), if not we re-call the AMControl.start().

Lectures des Sensor: At the “timer fired” event, call a read for the temperature, and at the “temperature read done”, call a read for the humidity. Wait for the “humidity read done” event, and then begin to construct the packet and to send it as explained below.

Do not forget to transform the 16-bit value temperature, to the temperature in degree and the true percentage of humidity. At the same time, you can put the integer and fractional parts of the temperature in the corresponding field of your message structure:

And to construct the structure:

```
Temp = 0.01*(double)val – 39.6;
T_integ = temp;
T_frac = (temp – T_integ)*100+0.5;
```

The same thing for the humidity as

```
Temp = -2.0468+val*(0.0367 – 1.5966e-6*val);
humid = Temp;
```

T_integ, T_frac and humid are global variables of type uint8_t that you should declare at the top of your module to hold your readings.

Packet preparation: To prepare the packet to be sent, we use the interface AMSenderC.AMSend with the related function getPayload(). Therefore, we add the components AMSenderC at the configuration file as:

```
components new AMSenderC(6);
```

Note: the “6” in the parentheses define the group address. It means that if the radio device receives a packet with another group number, the packet will not be delivered to the microprocessor (filtered out). So in this lab we fix the group address to 6 (an arbitrary value). Therefore, it allows several ZigBee networks to work in the same environment without confusion. Do the connection as:

```
App.AMSend -> AMSenderC.AMSend;
```

Do not forget to add in the module:

```
uses interface AMSend;
```

The packet that will be sent by the radio contains your message (the MyMsg structure), but also some headings. The type of the whole packet is defined as “message_t”. The type message_t is a structure with four fields as header, data, footer, and metadata. So we define a global variable named pkt as

```
messag_t pkt;
```

Now we can call the function getPayload() and give to it the size of our message (using `sizeof(MyMsg_t)`). The function returns us a pointer that points to the payload, where we can copy our message structure. The getPayload function also makes allocate memory for the entire packet and return in the argument the pointer that points to the whole packet:

```
MyMsg = call AMSend.getPayload(&pkt, sizeof(MyMsg_t));
```

The MyMsg should be declared as global in the module as

```
MyMsg_t * MyPtr;
```

Now that we have our pointer, we fill it and then we send it:

```
MyMsg -> NodeId = TOS_NODE_ID; // this value is fixed at the compilation
MyMsg -> PktNo = count++;
MyMsg -> Temp_int = T_integ;
MyMsg -> Temp_frac = T_frac;
MyMsg -> Humid = humid;
```

Note: The TOS_NODE_ID default value is 1. In order to change it to our own node ID, say "n", you can initialize it at compilation using

```
Make telosb install.n
```

Note that we used an 8-bit variable “count” that indicates the message number. So, we can find out at the reception if a message is lost or not.

Sending the packet: It is time now to send the packet. For that, we need to call the function send() using the interface AMSender of the component AMSenderC.

```
call AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(MyMsg_t);
```

The `AM_BROADCAST_ADDR` is a constant (x”FFFF”) indicating that the message is intended to all the motes in the group. So, at the receiver side, a packet with this value as the address, will be delivered by the radio to the µP with a proper event. If you want to send to a specific mote, you may put the ID of your destination instead of x”FFFF”.

This function returns a feedback (error) value to signal whether everything is ok or not. When the message is completely sent, the event “sendDone” is generated.

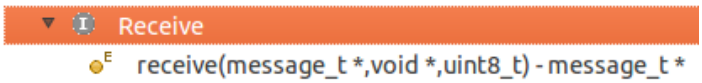
The programmer must pay attention not to send another packet while the sendDone is not received. A flag (busy) of type bool can be defined and initialized to FALSE. When the send() function is called, we switch the busy flag to TRUE and when the sendDone is received, we re-switch it to FALSE. We check this flag before calling the send() function. The sendDone function returns a pointer to the message that was just sent by radio. It can be used if you have several types of messages and you want to know this particular sendDone is the acknowledgment of which message. One can use the following lines:

```
event void AMSend.sendDone(message_t* msg, error_t error) {
        if (&pkt == msg) {
                busy = FALSE;
        }
 }
```

Objectives: Send a packet every 8 seconds using broadcast address, and containing your ID and the counter. Each time a sendDone is generated, print on the screen a message (for example "send done" and the value of the counter) to ensure that you are correctly sending packets. A receiver will check the message on the air that can validate your program.

7) Receiving a packet

Each time a packet is received by radio (if the group number is the same as you, and you are the destination of the packet, or if it is just a broadcast) an event is generated. Here, we add to the previous program the ability to receive packets also. In general we need the interface "Receive" with its event handler Receive.receive().



The component that implements the Receive interface is "AMReceiverC". Put in the configuration:

```
components new AMReceiverC(6);
```

Note that you should use the same group address as the transmitter. The line to be typed is

```
App.Receive -> AMReceiverC.Receive;
```

And the interface to be added in the module:

```
uses interface Receive;
```

At the reception of a packet, the handler Receive.receive is called:

```
event message_t * Receive.receive(message_t *msg, void *payload, uint8_t len)
```

This function returns a pointer, that points to the whole packet (msg). It also returns in the argument a pointer, that points to the payload. We must cast its type to the same structure that is created in both TX and RX sides. You may type the following lines in the handler:

```
RecMsg = (MyMsg_t *) payload;
printf("Msg No %d from (%d)  T=%d.%d H=%d\r\n", RecMsg->PktNo,
RecMsg->NodeID, RecMsg->Temp_int, RecMsg->Temp_frac, RecMsg->Humid);
return msg;
```

You should be able to monitor on your PC all the messages in air.

If you do not know the data structure (for a general sniffer for example), you may just cast to a pointer uint8_t and print everything in the data, as:

```
ptr = (uint8_t *) payload;
for (i=0; i<len; i++)
        printf("%d ",*ptr ++);
printf("\r\n");
return msg;
```

8) Flooding protocol

Here we implement the flooding protocol. We assume that the sink node's ID is 0, and the other ID's are from 1 to 10.

Algorithm's principals:

1- Each node sends its data once the button is pressed
2- On the receiving of a packet, each node repeat the packet if
   a. The node has not already repeated the same packet (same Id, same packet number)
   b. It is not its own packet repeated by another node

The structure of the payload is as follows

```
Typedef struct MyMsg_t{
      nx_uint8_t  SrcId;
      nx_uint8_t  OrgSrcId;
      nx_uint8_t  HopNo;
      nx_uint8_t  PckId;
      nx_uint8_t  Msg[20];
} MyMsg_t;
```

When you send your own packet

- the SrcId and OrgSrcId are the same, it is your own Id,
- The HopNo is one.

When you are repeating the packet of another node

- the SrcId is the yours and the OrgSrcId is the Id of the original src,
- The HopNo that you send will be what you received plus one.

You have also the possibility to add a text message of a maximum size of 20.

Hint: in order to put the string in the packet you may use the following trick:

```
strcpy((char*) MyPacket->Msg,"My Name");
```

Create a new project that you call “flooding”. Create the module “floodingC”, the configuration file “floodingAppC”, the header file “flooding.h” and the Makefile.

Go to the site of the lab and copy the text of all the four files and paste in the corresponding files.

Try to understand the logic of the program.

 Exercise to return to the teacher in **one week**: Explain how the program works (max 3 pages, avoid to copy and paste the code, you may use flowchart to explain).