Introduction to TinyOS & nesC programming (I)

Applications provided with TinyOS

TinyOS comes preloaded with many applications that will help you use your sensor node and understand how programming in TinyOS works.
You can find these programs in /opt/tinyos-2.x/apps/ and /opt/tinyos-2.x/apps/tutorials
The rang of application differs from a simple code that just starts the sensor node and stay idle to more sophisticated applications that uses the sensors and radio of the sensor node.

Programming your sensor for the first time

The sensor nodes that we will be using in this lab are TelosB by crossbow referred to as mote from now on.
The mote has a USB port that is connected to an internal UART controller. Windows needs a driver to be able to interact with the UART port. If you have not installed the driver on your computer please refer to the previous lab.

Follow these steps to program your mote

1. Connect your mote to a USB port on your PC.
2. Start cygwin and type *motelist* this will show you a list of motes connected to your PC with their COM port number.
3. We will try to install the Blink application located in the apps folder shown above
4. Navigate to apps/Blink
5. Type *make telosb*. This will compile the nesC code for the telosb platform and create a code image that will be installed later.
6. Type *make telosb* reinstall. And wait until the process is finished.
7. You should now notice that the LEDs started blinking on your mote you have successfully programmed your mote.

TinyOS programming model

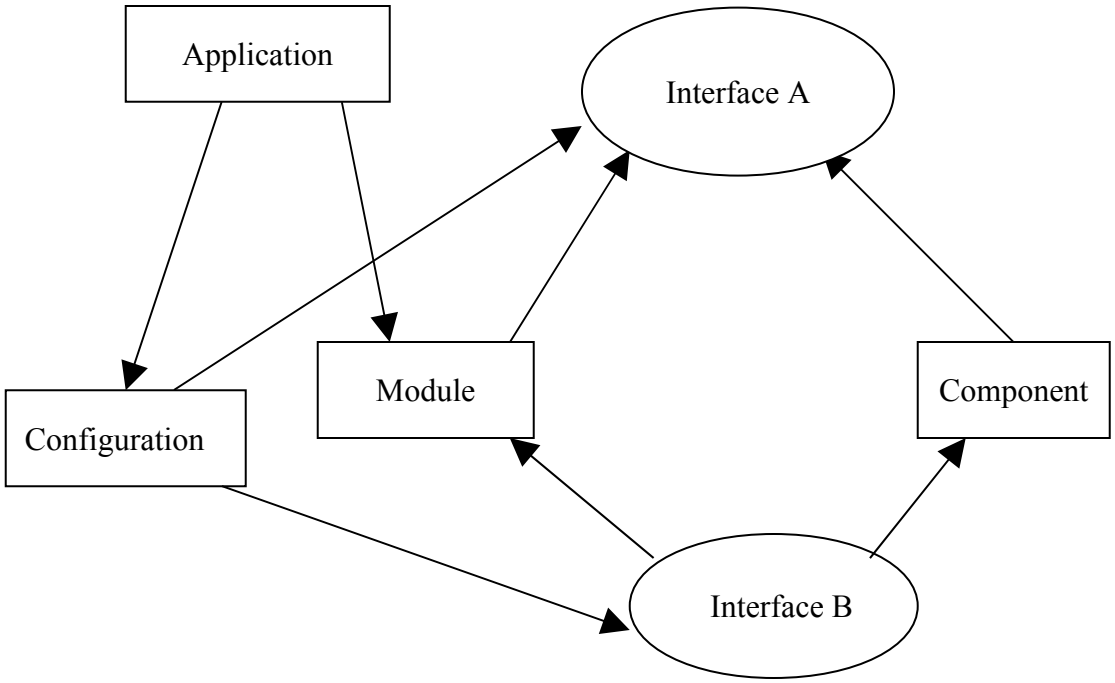
In nesC programming there are two fundamental elements described below

- **Interfaces**: nesC uses bi-directional interfaces that are used to communicate two components together. Interfaces only contain the method signature and not the implementation. A component can either provides an interface or uses an interface.
- **Components** cab be either one of two:
    - Configuration: a higher level description of the wiring of different components together.
    - Modules: which is the functional and actual code of the program you are writing. Mostly contains C code.

An Application usually has three files in its folder

- A configuration component.
- A module component.
- A Makerules file.

The Makerules file is used to tell the compiler where to look for component and interfaces and to load the needed parameter for the compiler. We will not discuss the Makerules file in this session.



The diagram below should make the idea clearer

File Naming Convention

There is an agreed on convention for naming the files in TinyOS between developers. It specifies how to name each of these elements

1. Modules are named xxxC.nc.
2. Configuration are name xxxAppC.nc.
3. Private module xxxP.nc should not be wired to directly.
4. Makerules always has the same name and has no extension.
5. Header files are named xxx.h from (C).

Where xxx is the name of your application.
For more information refer to TEP3 in tinyos.net

Revisiting the Blink application

You can find the blink application in /opt/tinyos-2.x/apps/blink
First let's look at the configuration file BlinkAppC.nc

```
configuration BlinkAppC  // Declaring a configuration file with file name.

 {
    // This where we states the interfaces that are used or provided  by this configuration.
 }
```

```
Implementation // this the implementation block
{
 components MainC, BlinkC, LedsC; // components that we are using
                                  // note BlinkC !!
 components new TimerMilliC() as Timer0; // different instances of timer
 components new TimerMilliC() as Timer1;
 components new TimerMilliC() as Timer2;


 BlinkC -> MainC.Boot;  //wiring Boot in BlinkC to MainC.Boot

 BlinkC.Timer0 -> Timer0; //Wiring timer0 interface used in BlinkC
 BlinkC.Timer1 -> Timer1;
 BlinkC.Timer2 -> Timer2;
 BlinkC.Leds -> LedsC; // wiring the Leds interface used in BlinkC to the LedsC component
}
```

Now let's take a look at the module BlinkC.nc

```
#include "Timer.h" // including a header file

module BlinkC  // Declaring the module with the file name
{
  uses interface Timer<TMilli> as Timer0; // using 3 instance of a parameterized interface
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds; // the interface Leds that will connect us to the LedsC component
  uses interface Boot; // the interface Boot that will connect us to the MainC component
}
```

Now let's take a look at the implementation part

```
implementation
{
  event void Boot.booted() // the booting sequence is finished and the event is signaled by MainC
  {
    call Timer0.startPeriodic( 250 ); //calling the command startPeriodic of the Timer Interface
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()// this event will be signaled by the timer when it finishes
  {
    call Leds.led0Toggle(); // calling led0Toggle of Leds Interface
  }

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

Commands and events

Functions that can be accessed through interfaces are either commands or events

- **Commands:** (A.K.A up calls) Are usually a set of instructions to do something like setting the timer or controlling the LEDs.
- **Events:** (A.K.A Down calls or callbacks) Are functions that are signaled when the command is finished. Hence event driven programming. If command has a corresponding event it must be implemented other wise the application will not compile correctly.