**Objective:**

We developan example application (TestSE1000Events) written for TinyOS that allows the use of the sensors in the [SE1000](https://www.advanticsys.com/wiki/index.php?title=SE1000) sensor board. This simple program makes use of auxiliary components apart from the main application, and as such is a good example to see how to organize your code. The idea behind this application is to transmit messages only when there is a state change in any of the sensors of the sensor board. One of the main challenges in wireless sensor networks is the energy consumption. By doing this we save battery as the radio is turned off while there are no messages to send.

Codes Access: Download the codes for the TestSE1000 from the link below:

<http://www.advanticsys.com/shop/documents/1352805730_TestSE1000Events.zip>

Place the code in your Tinyos/apps and make sure that the makefile has proper path to Tinyos directory (check the blink app for reference).

Follow the procedure step by step and prepare a final report including the received raw data and send it to lab instructor(s).

**Hardware**

The boards that we will use in this lesson are:

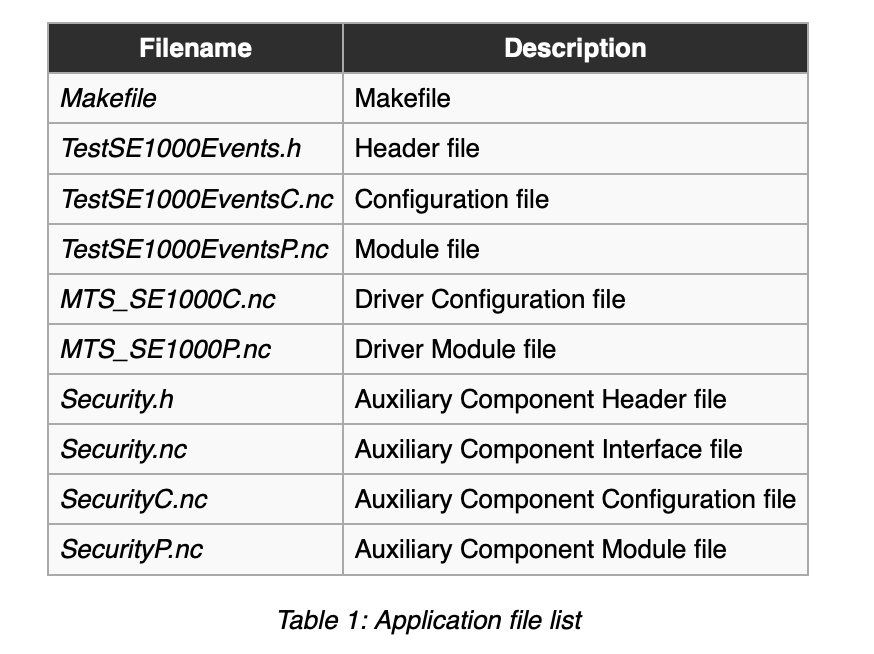
CM3000 (gateway)

SE1000 (sensor)

USB1000 (interface)

**Code Overview**

As with any TinyOS application, four main files are used. In this particular case, six additional files have been added, two of them being the raw sensor board driver, and the other four the auxiliary component that will be in charge of generating the events from the raw values obtained from the sensors:



### Makefile

The **Makefile**points to the public Configuration file TestSE1000EventsC, and also sets some of the available CC2420 flags. In this case, the radio is used without acknowledgements, and the channel selected is channel 26, which corresponds to 2330 MHz:

### TestSE1000Events.h

Header files such as **TestSE1000Events.h**can be used as auxiliary files to define constants and type structures, which a comfortable way of quickly modifying applications:

Contrary to other code examples, in this header file we only define the radio id of the transmitted messages and the structure of the sent packet.

### TestSE1000EventsC.nc

The Configuration file, **TestSE1000EventsC.nc**, links the different components used, such as:

* *TestSE1000EventsP*as the main module file
* *ActiveMessageC*for the Radio
* *AMSenderC*to handle message sending, and also to place the identification of the sent message thanks to the previously defined *TestSE1000Events\_AM\_ID*
* *SecurityC*for the auxiliary component

### TestSE1000EventsP.nc

**TestSE1000EventsP.nc**is the Module, or private part of the component. Here is were the main application is coded. In this basic example, after the *Boot*event takes place, the Security interface is called, via the *init()* command. From then on the auxiliary component will start generating events, that will be handled in the *stateChanged(uint8\_t pirstate, uint8\_t magstate , uint8\_t micstate, uint16\_t vref)* event. Once the new state is captured, the radio is started and the gathered message is broadcast to all listening motes. After the message is sent successfully, the radio is turned off to reduce power consumption.

### Auxiliary Component: Security

So, how are the events generated? Basically by sampling the raw sensors and analyzing the raw data inside the mote. To do so we have created the component **Security**, that has the following files:

#### Security.h

In **Security.h**the threshold and sample time for the different sensors are defined:

#### Security.nc

First of all we take a look at *Security.nc*. This is the interface file of the component, where we define our custom commands and events that will be implemented:  
In this case only one event and one command have been defined.

#### SecurityC.nc

The component's configuration file is *SecurityC.nc*. Apart from the components used, we must also indicate which interfaces will be provided and where is their implementation. To do so, in the specification area we declare that we provide the interface Security and in the implementation we wire it to the module file **SecurityP** by means of the '=' symbol. Notice that contrary to the main application, in this component we are not wiring the **Boot** interface of the MainC component.

In this case we are wiring the different **Read** interfaces (**PIR, MAG, MIC**) to those in the raw sensor board driver (MTS\_SE1000C), that basically sample the microcontroller's ADC ports.

#### SecurityP.nc

Finally, the module file *SecurityP.nc* is in charge of sampling the ADC ports and generating the event specified in the *Security*interface. The command **init()**performs an initial run through all the sensors and extracts the initial state of all of them. Once generated the corresponding event, the different timers for each sensor are launched, and whenever they fire they sill check for changes in the stored state of the sensors and signal the event **stateChanged** whenever that happens.   
  
Notice how the algorithms for the microphone and the magnetic sensor are quite simple, being reduced to check if the adc value is above or below the defined threshold. The PIR however has a counter to check a couple of times before assuming that the state has changed. This greatly reduce the number of false alarms.

### Raw Sensor Board Driver

**MTS\_SE1000C.nc** and **MTS\_SE1000P.nc** are the configuration and module files for the sensor board. They basically configure the ADC ports and provide the **Read** interfaces to get this raw count from the microcontroller's ADC.

**Installation:**

As with any other TinyOS applications, all the previous files have to be placed in the same folder, in a computer with a working TinyOS 2.x environment. Once that is done, we must compile the application. To do so open a console at the folder's location and type make telosb.

To install the compiled binary it is first needed to know the port at which the mote is mapped. In this example we are using a CM3000, together with a [USB1000](https://www.advanticsys.com/wiki/index.php?title=USB1000). To find out the port type motelist.

Whenever we use a mote plugged with a [USB1000](https://www.advanticsys.com/wiki/index.php?title=USB1000) the device description always shows it as the connected device, no matter which mote is being used. To install the binary, type make telosb reinstall.0x0046 bsl,/dev/ttyUSB0 where 0x0046 indicates the mote's id (in this case, 0x0046 in hex format) and /dev/ttyUSB0 the port where the mote is connected. Mote id is therefore configurable, and accepts any value between 0x0000 and 0xFFFF, that is any value between 0 and 65535.

**Listening to the data:**

Herein, we use another gateway (CM3000) to listen to incoming traffic.

**Step 1:** Download Tinyos release from github. You need to search for the SDK in it. Copy the SDK folder to your tinyos/support

**Step2:** Modify CLASSPATH in bashrc to

CLASSPATH=$CLASSPATH:$TOSROOT/support/sdk/java/tinyos.jar

**Step3:** Compile the apps/Basestation application to the gateway.

**Step4:** Use the following command to receive the raw data:

java net.tinyos.tools.Listen -comm serial@/dev/ttyUSB0:telosb

Once you got the raw data, make a snapchat and include it in your report.