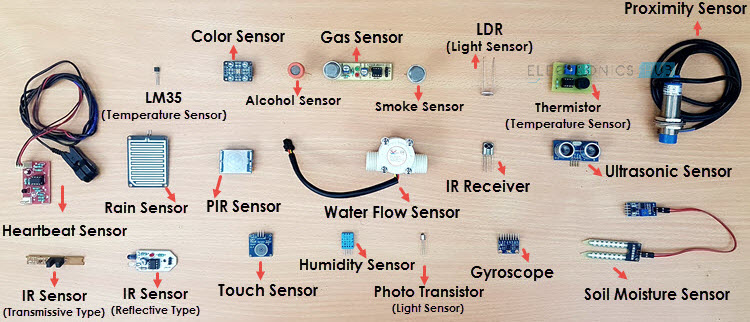
**Practical No. 1**

**Aim** : Understanding the Sensor Node Hardware. (For Eg. Sensors, Nodes(Sensor mote), Base Station,Graphical User Interface.)

**Program** :

1. Components :

* Sensors :



"Sensors" refer to devices or instruments that detect and measure physical properties or changes in the environment and convert this information into signals or data that can be interpreted, displayed, or used for various purposes. Sensors are crucial components in a wide range of applications, from industrial and scientific settings to everyday consumer electronics. Here are some common types of sensors: Temperature Sensor, Proximity Sensor, Accelerometer, IR Sensor (Infrared Sensor), Pressure Sensor, Light Sensor, Ultrasonic Sensor

* Nodes :



A sensor node (also known as a mote in North America), consists of an ****individual node from a sensor network**** that is capable of performing a desired action such as gathering, processing or communicating information with other connected nodes in a network.

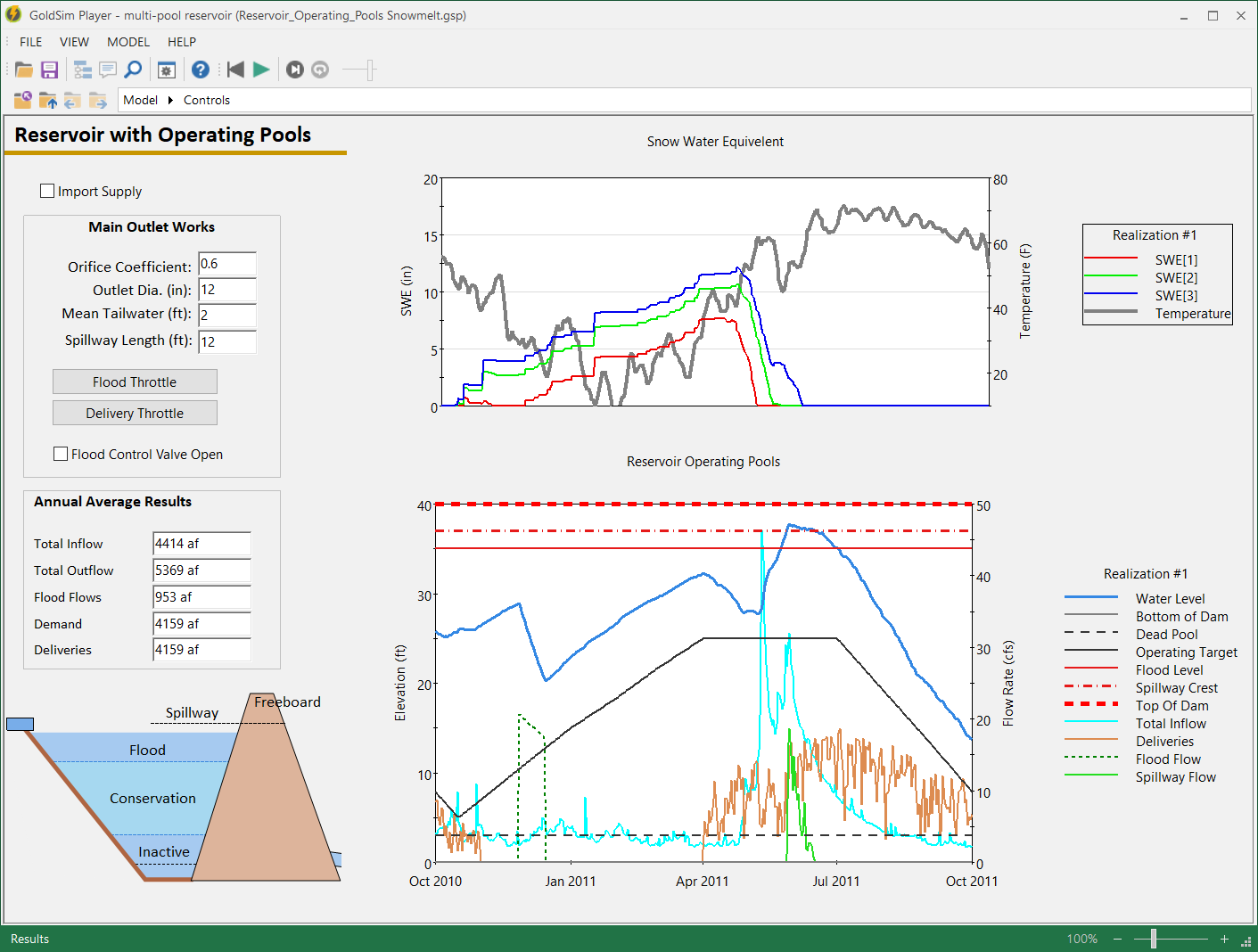
* Base Station :

A base station serves as a central connection point for a wireless device to communicate. It further connects the device to other networks or devices, usually through dedicated high bandwidth wire or fiber optic connections. Base stations are generally a [transceiver](https://www.techtarget.com/searchnetworking/definition/transceiver), capable of sending and receiving wireless signals; otherwise, if they only transmitted signals out, they would be considered a transmitter or broadcast point. A base station will have one or more radio frequency ([RF](https://www.techtarget.com/searchnetworking/definition/radio-frequency)) antennas to transmit and receive RF signals to other devices.

* Graphical User Interface :

GUI is the interface that uses graphical elements to let people interact as per requirement with electronic devices including computers, laptops, tablets, and smartphones. In terms of human-computer interaction systems or technology, it’s a very important component of software application programming since it substitutes actions for the text-based commands in the system. Whether it’s a text file, object, image, or video as per requirement, it displays all types of required content a user could envision in the system. Additionally, it can be featured in the gaming platform where the resolution is visible or optimal.

Eg : GUI of GoldSim



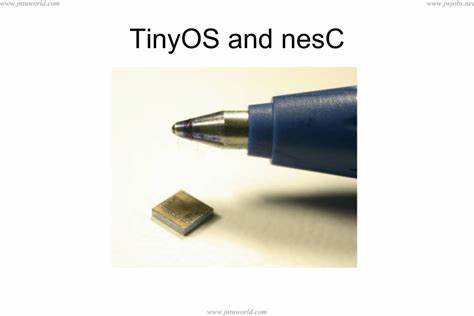
Conclusion :

Practical No. 2

Aim : Exploring and understanding TinyOS computational concepts: - Events, Commands and Task.

1. nesC model
2. nesC Components

Procedure :



**TinyOs :**

* **Component-based architecture**
* Reusable system components: ADC, Timer, Radio
* **Tasks and event-based concurrency**
* No user-space or context switching supported by hardware
* Tasks run to completion only preempted by interrupts
* **All long-latency operations are *split-phase***
* Operation request and completion are separate functions

**Introducing nesC :**

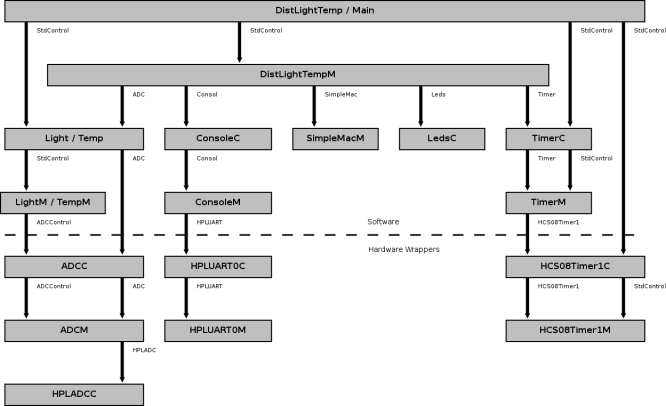
* A “holistic” approach to networked embedded systems
* Supports and reflects TinyOS's design
* Extends a subset of C
* A static language
* All resources known at compile-time
* Call-graph fully known at compile-time

**Design Decisions for nesC :**

* Components
* Bidirectional interfaces
* Simple expressive concurrency model
* Whole-program analysis

**Components :**

* Challenge: platform diversity, flexible SW/HW boundary, applications deeply tied to hardware
* Encourages modular design
* Restrict access to private data
* Allow underlying implementations to be replaced easily
* Can abstract HW using thin wrappers
* Allow specialization of applications to hardware



**Module Components :**

* Modules implement application code
* Modules have private state
* Sharing of data among components is discouraged
* Convention:
* Module names end with 'M', e.g. BlinkM
* Module variables start with 'm\_', e.g. m\_timers

**Configuration Components :**

* Configurations wire other components together
* All applications have a top-level configuration
* A component interface may be wired zero or more times
* Used for StdControl to implement power management
* Convention: Configuration names end with 'C', e.g. TimerC (unless it is the top-level configuration)

**/\* BlinkM.nc \*/**

**module** BlinkM {

**provides interface** StdControl **as** Control;

**uses interface** Timer;

**uses interface** Leds;

} **implementation** {

**command** result\_t Control.init() {

**call** Leds.init();

**return** SUCCESS;

}

**command** result\_t Control.start() { /\* ... \*/ }

**command** result\_t Control.stop() { /\* ... \*/ }

**event** result\_t Timer.fired() {

**call** Leds.redToggle();

**return** SUCCESS;

}

}

**/\* Blink.nc \*/**

**configuration** Blink {

} **implementation** {

/\* *Declare used components.* \*/

**components** Main, BlinkM, SingleTimer, LedsC;

/\* *Wire components together.* \*/

Main.StdControl -> SingleTimer.StdControl;

Main.StdControl -> BlinkM.StdControl;

BlinkM.Timer -> SingleTimer.Timer;

BlinkM.Leds -> LedsC;

}

**Interfaces**

**/\* Timer.nc \*/**

**includes** Timer; /\* *Include C types from Timer.h* \*/

**interface** Timer {

**command** result\_t start(char type, uint32\_t interval);

**command** result\_t stop();

**event** result\_t fired();

}

**/\* SyncAlarm.nc \*/**

**interface** SyncAlarm<Precision\_t> {

**command** result\_t armCountdown(Precision\_t timeout);

**command** result\_t armAlarmClock(Precision\_t time);

**command** result\_t stop();

**event** result\_t alarm();

}

**Parameterized Interfaces**

**module** TimerM {

**provides interface** Timer[uint8\_t id];

} **implementation** {

/\* ... \*/

Timer\_t m\_timers[NUM\_TIMERS];

**command** result\_t Timer.isSet[uint8\_t timer]() {

return m\_timers[timer].isset;

}

**task** void timerCheck() {

uint8\_t timer;

for (timer = 0; timer < NUM\_TIMERS; timer++)

if (m\_timers[timer].fired)

**signal** Timer.fired[timer]();

}

/\* ... \*/

}

**configuration** MyApp { /\* ... \*/ }

**implementation** {

**components** MyAppM, TimerC, /\* ... \*/;

MyAppM.SampleTimer -> TimerC.Timer[unique(“Timer”)];

}

**Tasks and Events**

**module** LightM {

/\* ... \*/

} **implementation** {

uint16\_t light\_data

**task** void processLightdata() {

uint16\_t local\_light\_data;

**atomic** local\_light\_data = light\_data;

/\* Process light data. \*/

**if** (!done)

**post** anotherTask()

}

**async event** result\_t Light.dataReady(uint16\_t data) {

**atomic** lightData = data;

**post** processLightData();

**return** SUCCESS;

}

**event** result\_t SensorTimer.fired() {

**return** call Light.getData();

}

}

**Practical No. 3**

**Aim** : Understanding TOSSIM for

* Mote-mote radio communication
* Mote-PC serial communication

**Procedure** :

* **TOSSIM for Mote-mote radio communication**
* TOSSIM is a discrete event simulator for TinyOS sensor networks. Instead of compiling a TinyOS application for a mote, users can compile it into the TOSSIM framework, which runs on a PC.
* This allows users to debug, test, and analyze algorithms in a controlled and repeatable environment.
* As TOSSIM runs on a PC, users can examine their TinyOS code using debuggers and other development tools.
* This document briefly describes the design philosophy of TOSSIM, its capabilities, its structure.
* It also provides a brief tutorial on how to use TOSSIM for testing or analysis.
* **Components**

A number of components implement the basic communications and active message interfaces.

* AMReceiverC - Provides the following interfaces: Receive, Packet, and AMPacket.
* AMSenderC - Provides AMSend, Packet, AMPacket, and PacketAcknowledgements as Acks.
* AMSnooperC - Provides Receive, Packet, and AMPacket.
* AMSnoopingReceiverC - Provides Receive, Packet, and AMPacket.
* ActiveMessageAddressC - Provides commands to get and set the node's active message address.
* This interface is not for general use and changing a node's active message address can break the
* network stack, so avoid using it unless you know what you are doing
* **Compiling TOSSIM**

TOSSIM is a TinyOS library. Its core code lives in tos/lib/tossim. Every TinyOS source directory has an optional sim subdirectory, which may contain simulation implementations of that package. For example, tos/chips/atm128/timer/sim contains TOSSIM implementations of some of the Atmega128 timer abstractions.

To compile TOSSIM, you pass the sim option to make:

$ cd apps/Blink

$ make micaz sim

* **Running TOSSIM with Python**

Go into the RadioCountToLeds application and build TOSSIM:

$ cd tinyos-2.x/apps/RadioCountToLeds

$ make micaz sim

We'll start with Python in interactive mode. To start the Python interpreter, type:

$ python

* **TOSSIM for Mote-PC radio communication :**

The basic abstraction for mote-PC communication is a **packet source**. A packet source is exactly that: a communication medium over which an application can receive packets from and send packets to a mote. Examples of packet sources include serial ports, TCP sockets, and the SerialForwarder tool. Most TinyOS communication tools take an optional -comm parameter, which allows you to specify the packet source as a string. For example:

$ java net.tinyos.tools.Listen -comm serial@COM1:telos

tells the Listen tool to use the COM1 serial port (on a Windows machine) at the correct speed for a telos mote, while

$ java net.tinyos.tools.Listen -comm serial@/dev/ttyS0:micaz

tells Listen to use the serial port /dev/ttyS0 (on a UNIX machine) at the correct speed for a micaz mote. The first step to testing your serial port is to install the apps/tests/TestSerial application on a mote. This application sends a packet to the serial port every second, and when it receives a packet over the serial port

Once you have installed TestSerial, you need to run the corresponding Java application that communicates with it over the serial port. This is built when you build the TinyOS application. From in the application directory, type:

$ java TestSerial

If you get a message like

The java class is not found: TestSerial

it means that you either haven't compiled the Java code (try running make *platform* again) or you don't have . (the current directory) in your Java CLASSPATH.

Because you haven't specified a packet source, TestSerial will fall back to a default, which is a SerialForwarder. Since you don't have a SerialForwarder running, TestSerial will exit, complaining that it can't connect to one. So let's specify the serial port as the source, using the -comm parameter as described above. The syntax for a serial port source is as follows:

serial@<PORT>:<SPEED>

PORT depends on your platform and where you have plugged the mote in. For Windows/Cygwin platforms, it is COM*N*, where *N* is the port number. For Linux/UNIX machines, it is /dev/ttyS*N* for a built-in serial port, or one of /dev/ttyUSB*N* or /dev/usb/tts/*N* for a serial-over-USB port. Additionally as we saw in lesson 1, on Linux you will typically need to make this serial port world

writeable. As superuser, execute the following command:

chmod 666 *serialport*

* **MOTECOM**

If you do not pass a -comm parameter, then tools will check the MOTECOM environment variable for a packet source, and if there is no MOTECOM, they default to a SerialForwarder. This means that if you're always communicating with a mote over your serial port, you can just set MOTECOM and no longer have

to specify the -comm parameter. For example:

export MOTECOM=serial@COM1:19200 # mica baud rate

export MOTECOM=serial@COM1:mica # mica baud rate, again

export MOTECOM=serial@COM2:mica2 # the mica2 baud rate, on a different serial port

export MOTECOM=serial@COM3:57600 # explicit mica2 baud rate

* **BaseStation and net.tinyos.tools.Listen**

BaseStation is a basic TinyOS utility application. It acts as a bridge between the serial port and radio network. When it receives a packet from the serial port, it transmits it on the radio; when it receives a packets over the radio, it transmits it to the serial port. Because TinyOS has a toolchain for generating and sending packets to a mote over a serial port, using a BaseStation allows PC tools to communicate directly with mote networks.

$ java net.tinyos.tools.Listen

Listen creates a packet source and just prints out every packet it sees. Your output should look something like this:

00 FF FF 00 00 04 22 06 00 02 00 01

00 FF FF 00 00 04 22 06 00 02 00 02

00 FF FF 00 00 04 22 06 00 02 00 03

00 FF FF 00 00 04 22 06 00 02 00 04

00 FF FF 00 00 04 22 06 00 02 00 05

00 FF FF 00 00 04 22 06 00 02 00 06

00 FF FF 00 00 04 22 06 00 02 00 07

00 FF FF 00 00 04 22 06 00 02 00 08

00 FF FF 00 00 04 22 06 00 02 00 09

00 FF FF 00 00 04 22 06 00 02 00 0A

00 FF FF 00 00 04 22 06 00 02 00 0B

typedef nx\_struct BlinkToRadioMsg {

nx\_uint16\_t nodeid;

nx\_uint16\_t counter;

} BlinkToRadioMsg;

**Conclusion:**