VirtualSense User Guide 2

Application Programming Interface

University of Urbino & NeuNet
www.virtualsense.it
Revision 1.0

April 3, 2014

This guide applies to VirtualSense hardware platform 1.1.0.

Contents

List of Figures

1 Introduction

The availability of off-the-shelf micro controller units based on energy efficient 16-bit RISC processors which provide a wide range of low-power inactive modes with average current in the range of micro Watts and wake-up times in the range of micro seconds makes it possible to develop ultra-low-power sensor nodes able to run a virtual machine to speedup the development and the deployment of sensing/monitoring applications.
VirtualSense is an open-hardware/open-source project which aims at the development of IEEE 802.15.4-compliant low-cost ultra-low-power wireless sensor nodes providing a Java-compatible runtime environment which grants to the programmer full control of the low-power states of the hardware.

2 General description and Key Features

VirtualSense is an ultra-low power wireless node for use in wireless sensor networks (WSNs) subject to tight power constraints. Thanks to the on board Java compatible virtual machine (VM) it allows programmers to rapidly develop monitoring applications and communication protocols. VirtualSense makes use of IEEE 802.15.4 wireless transceivers in order to standardize communication and to inter-operate with other existing devices. The set of on board sensors (including pressure, temperature, and light), together with the possibility to easily connect any external sensor/actuator, allows VirtualSense to be used in a wide range of application fields.
In order to promote research and development VirtualSense adopts an open-hardware/open-source model. In particular, it mounts widely available off-the-shelf components and it makes publicly available all PCB schematics. The open-source software stack is based on a modified version of Darjeeling java-compatible VM running on top of Contiki operating system.

3 Key components and features

The key components of VirtualSense 1.0 are listed below:

- 250kbps 2.4GHz IEEE 802.15.4 Texas Instruments cc2520 Wireless Transceiver
- 25MHz Texas Instruments MSP430f54xxa microcontroller unit (MCU) with 16k RAM and 128k Flash
- Integrated Humidity, Temperature, and Light sensors
- 512K I2C™ Serial EEPROM
- On-board 48-bit I2C Extended Unique Identifier (EUI-48™)
- On-board programmable ultra-low-power RTC

The distinguishing features include:

- Ultra low power consumption ($\approx$10W in hibernation, $\approx$100W in sleep mode, 50/60mW in send/receive modes, respectively)
- with state-of-the-art energy harvesting modules
- Fast wakeup from sleep mode ($<$5s)
- Programmable timed wake-up from any low-power mode
- Sensitivity to asynchronous external events
- Integrated 12-bit ADC/DAC
- Integrated Supply Voltage Supervisor (SVS)
- Integrated DMA Controller
- USB 2.0 RS232/UART communication with a PC
- Interoperability with other IEEE 802.15.4 devices
- Open-source software stack
- Contiki MAC-layer compatibility
- Java-compatible run-time environment
- Easy Over the Air (OTA) programming

4 Architecture

VirtualSense is made of ultra-low-power components in order to keep the average consumption compatible with state-of-the-art energy harvesters. Figure 2 shows the functional block diagram representing the node architecture. The core is a MCU belonging to the Texas Instrument MSP430F54xxa family. It communicates through I2C™ bus with a Microchip 24AA025E48 Extended Unique Identifier and with a Microchip 24AA512 serial 512K EEPROM. Using the SPI bus, the MCU manages the Texas Instruments CC2520 2.4GHz IEEE 802.15.4 RF transceiver and communicates with the NXP PCF2123 ultra low-power real time clock/calendar.

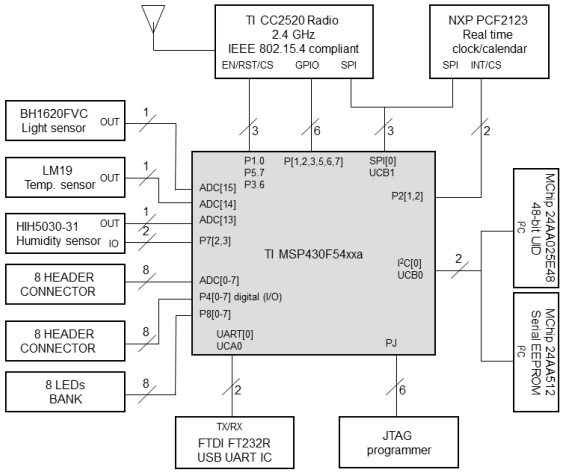


Figure 1: The functional block diagram

5 Hardware and software stacks

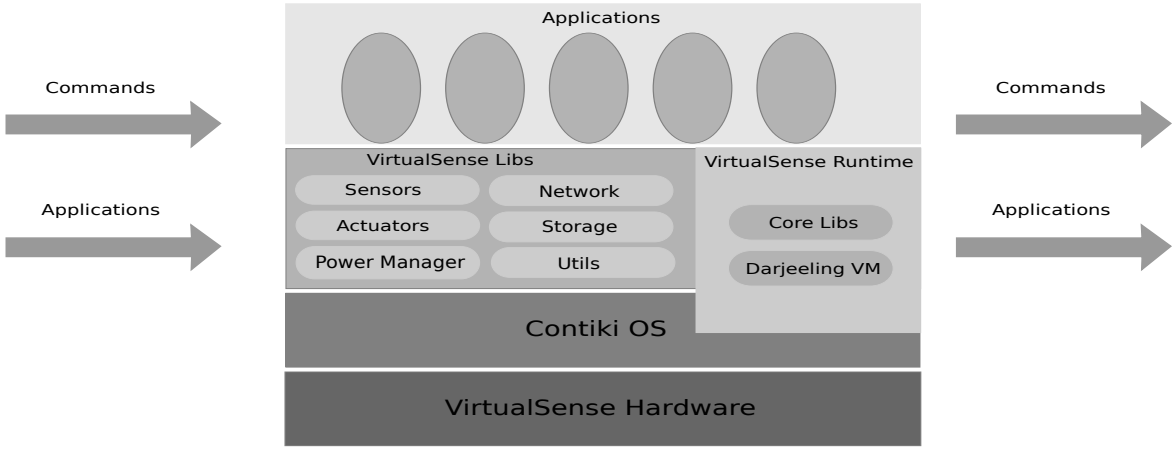


Figure 2: Hardware and software staks

At the lowest stack level there is VirtualSesnse hardware that provides to higher levels: an network interface, an ultra-low-power microcontroller and a set of sensors. In addition to ensuring low power consumption, the 20-bit processor on the VirtualSense, allows hight performance so that at the base of software stack of VirtualSense can be an operative system like CONTIKI. The great versatility of CONTIKI permit to run on a processor with 16k RAM and 128k Flash, a real virtual machines like the JVM that allow usage an hight-level programming language like Java™. The VirtualSenese Runtime represents the most important stack level, because provide to powered application leaving the approach near-hardware to using a simple set of API (described in VirtualSense Libs) that give access to all hardware functionality. The high abstraction of VirtualSense runtime architecture provide to create multi-threading applications that can run concurrently on a single node increasing the use cases of VirtualSense. Because the apps running on Virtualsense are not on the firmware but in a higher level, them can be removed, updated or added by remote only sending some special package and commands on the network. The hight performance and great versatility of VirtualSense not affect the power consumption, thanks to hight optimization of VirtualSense runtime architecture, that takes full advantage of the low-power state of MSP430 microcontrollers.

6 Install VirtualSense Development Kit

6.1 System Requirements

On the development of VSDK (VirtualSense Development Kit) we tried to make it lightest as possible to be portable on any machines. VSDK is tested on Linux OS that the follow requirements referring to Linux based environments.

Therefore the only requirements for use VSDK are:

- JDK (Java Development Kit).
- Ant compiler.
- srecord package used for manipulating EPROM load files.
- Eclipse IDE. Only if you want develop application using an integrated programming environment.

6.2 Installation

6.2.1 Install Requirements

In a linux OS open **Terminal** (Alt + Ctrl + T) and insert the following commands:

```
sudo apt-get install openjdk-7-jre ant srecord
```

If you want work with IDE insert also:

```
sudo apt-get install eclipse
```

6.2.2 Install VSDK

Download package **VirtualSense_SDK.1.1.0.zip** from http://www.virtualsense.it/download.

If you don't wont use Eclipse IDE unpack the downloaded package and skip next step. Otherwise don't unpack the package and perform next step.

```
~/home/virtualsense/Downloads$

unzip VirtualSense_SDK.1.1.0.zip -d /home/virtualsense
```

Open Eclipse and go on **File** > **Import**, select **Existing Projects into Workspace** and click **Next**.

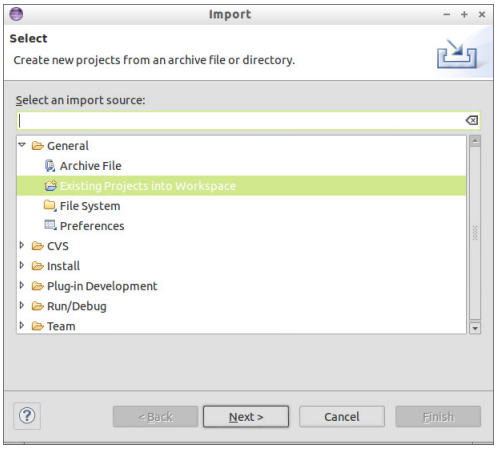


Figure 3: Import VSDK on Eclipse step1

Check **Select archive file**, browse VSDK package previous downloaded, uncheck all projects detect on Projects section except VirtualSense_SDK.1.1.0 and click **Finish**.

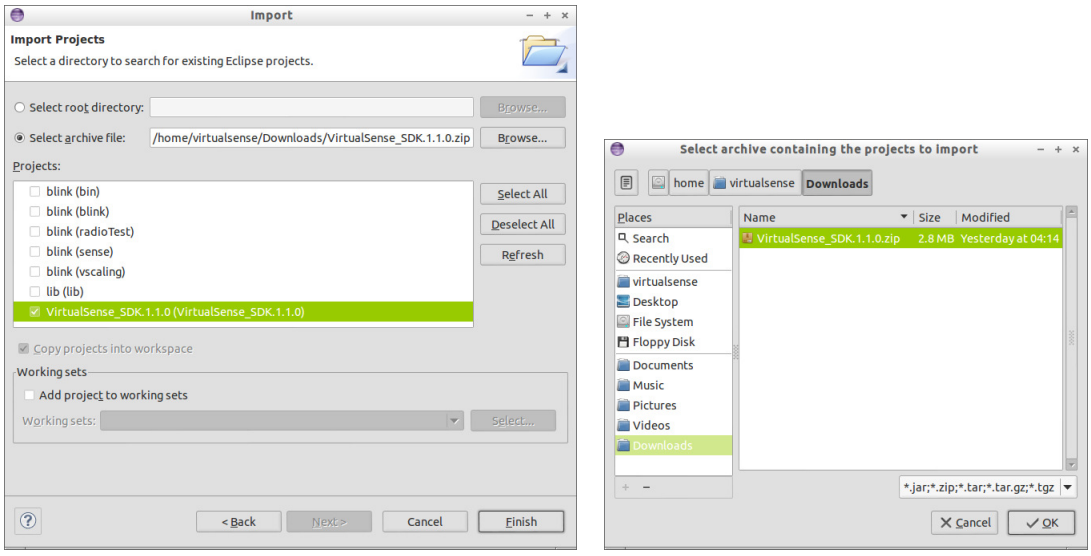


Figure 4: Import VSDK on Eclipse step2

6.2.3 Install VirtualSense-BSL

The last step is install **BSL** (Boot Strap Loader) **Programmer**. Download virtualsense-bsl from http://virtual-sense.googlecode.com/files/VirtualSenseBSL.tgz

Ensure that you have the necessary packages to compile programs that use **libusb**. On Debian or Ubuntu systems, you might need to do:

```
apt-get install libusb-dev
```

After that, unpack and compile the source code with:

```
tar xvfz VirtualSenseBSL.tar.gz
cd VirtualSenseBSL
make WITHOUT_READLINE=1
```

After compiling, install the binary by running (as root):

```
make install
```

7 Basic Application Structure

In order to develop an a general application the first step is write the source code, which is then compiled and finally executed. When we are going to write an application for VirtualSense, designed to work with Java source, the first step is define the Class that contains the **motemain** method. The method **motemain** is the first method executed on startup of VirtualSense applications, define the behavior of mote in question and must be define as:

```
public static void motemain()
{
  // TODO code application logic here
}
```

So the generally structure of a VirtualSense application named FirstApp must be as:

Listing 1: FirstApp.java

```
// Libraries
import <libraries>    // Import libraries to use here

/**
 * First application
 * @author virtualsense
 */
public class FirstApp
{
  /**
  * Definition of motemain method
  */
  public static void motemain()
  {
    // Statments for inizialize device

    // Behavior of application
    while(true)
    {

      // TODO code application logic here

    }
  }
}
```

The structure is similar as a normal java application, at the top are the libraries used, after there is the definition of class in witch we find the motemain definition. Motemain method is based on while-true cycle because this method define the complete behavior of mote and if it return the execution ends and mote will be restarted, so if you want create an application that run indefinitely or in base at the value of a condition you must insert a cycle.

8 VirtualSense API

When develop an application for a wireless sensors network is most important have a set of tool that provide to use all functionalities made available from the node. VirtualSense lib define a set of APIs that give you a full control of all functionality provided by VirtualSense mote. The following table explain a overview of all APIs contained of VirtualSense lib.

java.io	Java input and output classes.
java.lang	Java support.
java.util	Java utilities.
javax.virtualsense.actuators	Provides management of actuators on the board.
javax.virtualsnese.concurrent	Provides management of synchronized and race condition.
javax.virtualsnese.network	Provides management of network functionality.
javax.virtualsense.powermanagment	Provides management of power consumption of mote.
javax.virtualsense.sensors	Provides management of sensors on mote.

8.1 Actuators Interfaces

The only actuators available on VirtualSense motes are leds. More precisely on Radio Layer VirtualSense motes are equipped with 3 leds, so actuator interfaces consists of once class named **Led.java** that provide leds management.

8.1.1 Leds

The class **Led.java** is a native interface that defines only the method **setLed**.

Led
static final short **LED1**
static final short **LED2**
static final short **LED3**
native void **setLed**(int led, boolean state)

This class should not be instantiated but for control one led is simply import library **actuators.Leds** and invoke **Led.setLed** for change the state of every led.

```
import javax.virtualsense.actuators.Leds;
```

For control the state of every led the class provide also 3 constant attributes one for each Led. For example if you wont turn on led 2 you must use follow command:

```
Leds.setLed(Led.LED2, true);
```

8.2 Sensors Interfaces

The sensors interface manage all device that interfaces VirtualSense with outside. Precisely VirtualSense has:

- set of Analog-Digital Converter.
- brightness sensor.
- pressure sensor.
- temperature sensor.

For use one of them is sufficient import library **Sensors** with command:

```
import javax.virtualsense.sensors.*;
```

and use a corresponding class.

8.2.1 ADC

The class that manages the set of analog to digital converters is **ADC.java**.

ADC
static final Byte **CHNL1**
static final Byte **CHNL2**
static final Byte **CHNL3**
static final Byte **CHNL4**
static final Byte **CHNL5**
static final Byte **CHNL6**
static final Byte **CHNL7**
static final Byte **CHNL8**
native short **init**(channels)
native short **getNrADCs**()
native short **read**(channel)

VirtualSense motes has a set of N ADCs mapped on pins which can be used as several function. For enable one ADC you must invoke method **init** passing a **byte** value witch refer to ADC to select.

All various ADC channels are down when not use because consume more power, then in order to **save power** enable only you use. For example if you want enable the channels 2 and 3 of ADC you must use follow command:

```
ADC.init(CHNL1 + CHNL2);
```

All available channel is masked with an constant attribute provided by class ADC, refer to schematic for get pin-out configuration.
To read level of an ADC channel class ADC provide method **read** that return level read from ADC making an average of several samples. For read value of channels 1 and 2:

```
short level1 = ADC.read(CHNL1);
short level2 = ADC.read(CHNL2);
```

8.2.2 Light

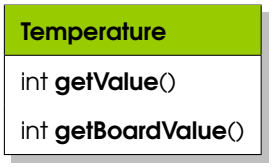
The class that manages brightness sensor is **Light.java**.

The Light class provide the method **getValue()** for read the current brightness level, this level is mapped on an integer value where 2147483647 is assumed as full light and -2147483648 is full darkness. For read the current brightness level you must use:

```
int lightLevel = Light.getValue();
```

8.2.3 Temperature

The class that manages a Temperature sensor is **Temperature.java**.



The class Temperature read two different level of temperature: cpu temp (using an internal ADC) and external temperature using the on board sensor. both temperature levels are mapped on an integer value than. For read current temperature level on both sensors the commands are:

```
int boardTemp = Temperature.getValue();
int extTemp = Temperature.getValue();
```

8.2.4 Pressure

The class that manages a Pressure sensor is **Presure.java**.

The Light class provide the method **getValue()** for read current pressure level measured whit barometer sensor. Pressure value is mapped on an integer than. For read current pressure level command is:

```
int pressure = Pressure.getValue();
```

8.3 PowerManagement Interfaces

The power management interface provide a set of functionality for manage the power consumption of VirtualSense motes. For use PowerManagment functionality you must import the library:

```
import javax.virtualsense.powermanagement;
```

In detail using this interface you can:

- Set the working frequency of a mote.
- Put mote in Standby.
- Put mote in DeepSleep.
- Put mote in SystemHibernation.

PowerManager
static final short **MCU_4MHZ**
static final short **MCU_8MHZ**
static final short **MCU_12MHZ**
static final short **MCU_16MHZ**
static final short **MCU_20MHZ**
static final short **MCU_25MHZ**
native void **setMCUFrequency**(short freq)
native void **setSystemClockMillis**(millis)
native void **slowDownClockByFactor**(factor)
native void **standby**()
native void **deepSleep**()
native void **systemHibernation**()
native void **scheduleRTCInterruptAfter**(int minutes)
void **systemHibernation**(int minutes)
void **deepSleep**(int minutes)

Class PowerManager define a set of constant attributes to set the working frequency of MCU in a VirtualSense mote. The operating frequencies of VirtualSense's MCU are 4MHZ, 8MHZ, 12MHZ, 16MHZ, 20MHZ, 25MHZ. For set a working frequency you must invoke method **setMCUFrequency** passing as parameter a frequency constant. For example to set working frequency to 16MHz you must invoke:

```
PowerManager.setMCUFrequency(PowerManager.MCU_16MHZ);
```

The rest of methods define by PowerManager are use for manage power consumption:

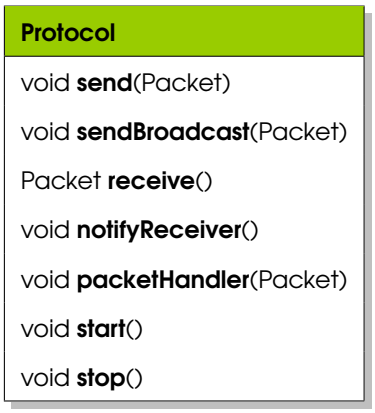
- **setSystemClockMillis(millis)** Sets the system clock at the specified millisecond. When you try to set the system clock the MCU is put in LPM3 mode.
- **slowDownClockByFactor(factor)**
- **standby()** Puts the MCU in LPM3 where consume approximatively 2.1 uA. In this mode wakeup is possible through all enabled interrupts.
- **deepSleep()** Puts the MCU in LPM4 where consume approximatively 1.3 uA. In this mode wakeup is possible through all enabled interrupts.
- **systemHibernation()** Puts the MCU in LPM4.5 where consume approximatively 0.1 uA. To wake up from this mode the system need to restart by an interrupt over P1 or P2 port. Before is put to LPM4.5 MCU write machine state on non-volatile memory. When resume MCU check if exist an hibernation state and if exist the execution of the VM will restart from the subsequent statement. Otherwise the execution will restart from main.
- **scheduleRTCInterruptAfter(minutes)**
- **systemHibernation(minutes)** Hibernates the system for the specified time. The MCU can be woke up by un interrupt over P1 or P2 port or at the end of specified time.
- **deepSleep(minutes)** Puts the MCU in LPM4 for the specified time. MCU can be woke up through all enabled interrupts or at the end of specified time.

8.4 Network Interfaces

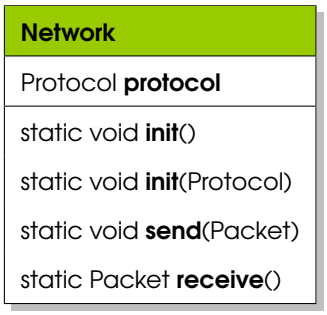
The Network interface manage the communication between nodes in a VirtualSense sensor network. Precisely manges the radio layer on VirtualSense for: create a network between notes, exchange some information within the network, route packet into the network using a given protocol. The classes that provide that functionality are: Network, Protocol, Packet and NullProtocol. For add network functionality to an application you must include follow library.

```
import javax.virtualsense.network.*;
```

To create a VirtualSense network the first step is extends abstract class **Protocol** that permits to define the system management of packets exchanged between nodes. Class **Protocol** can not be used directly within an application because on running is represented by a separate thread that listens on radio channel and manages all incoming packets.

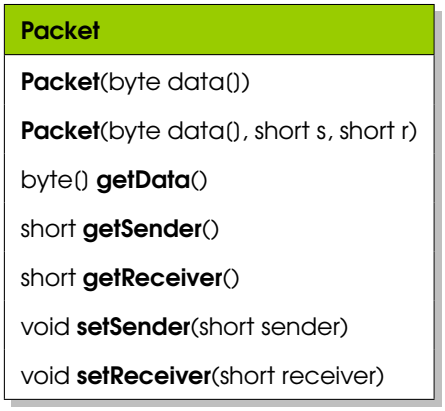


The method that must be define when creates a Protocol for VirtualSense is **packetHandler**. This method is invoke, by Protocol thread, whenever there is an incoming packet. If the incoming packet must be forwarded to application you must invoke the method **notifyRecever** which will indicate to the application that there is a packet for her. The class to be used directly in the application for manage network functionality **Network**.



Network class, which should not be instantiated, once initialized, with method **init**, creates and start thread protocol using class passed as a parameter. If you don't want extend class Protocol you can use a **NullProtocol** but in this way all incoming packet are froward at application threads which are responsible for the management. Network class provide for exchange packets with others mote two methods **send** and **receive**. **Send** method sends a **Packet** type message follow rules required by the Protocol. **Receive** method blocking the invoker to wait a message and when one is coming return an **Packet** type object.

All packets exchanged on a VirtualSense network are of following type.



Packet type contains: sender information, receiver information and the content of the packet expressed as byte array.

8.4.1 Networking example

Every time is created an network application on VirtualSense, there are almost two thread running on every mote. Main thread and Protocol thread.

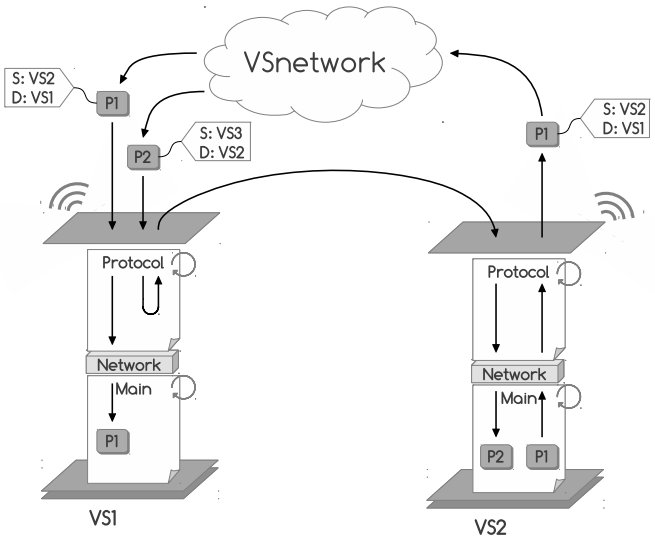


Figure 5: VirualSense networking

Main thread performs processing and interchanges information on network through Protocol thread. Protocol thread manage all packet incoming on mote passing to main thread those destined to it and forward those intended for other nodes. Two threads communicate by Network type object that provide method for send and receive packets.

9 VirtualSense Example

VSDK provide various sample applications useful for getting started to develop application. The example are contained in:

`VirtualSense_SDK.1.1.0/src`

9.1 Hello World example

Example of Hello world on VirtualSense.

Listing 2: HelloWorld.java

```
// Libraries
import javax.virtualsense.VirtualSense;


public class HelloWorld
{
  public static void motemain()
  {
    while(true)
    {
      // Write the String "Hello World!!" on serial port
      System.out.println("Hello World!!");

      Thread.sleep(1000);
    }
  }
}
```

Write string "Hello World!!" on serial port every second.

9.2 Blink example

Simple example of blink VirtualSense application.

Listing 3: Blink.java

```
// Libraries
import javax.virtualsense.actuators.Leds;
import javax.virtualsense.powermanagement.PowerManager;
import java.lang.Runtime;

public class Blink
{
  public static void motemain()
  {
    boolean state = true; // State of leds

    /* Slow down the system clock (normally it is configured at 10 ms) to
       reduce power consumption leaves the CPU in the LPM3 state */
    PowerManager.setSystemClockMillis(500);

    while (true)
    {
      // Toggles sequentially leds state every second
      for (short i = 0; i < 2; i++)
      {
        Leds.setLed(i, state);
        Thread.sleep(1000);
      }
      state = !state;
    }

  }
}
```

Blink application after sets clock at 500 millisecond, update every second the state of all leds on mote, sequentially starting from first to last.

9.3 Sense example

Sensing application on VirtualSense on board sensors.

Listing 4: Sense.java

```
// Libraries
import javax.virtualsense.actuators.Leds;
import javax.virtualsense.sensors.*;
import javax.virtualsense.VirtualSense;
import javax.darjeeling.Darjeeling;

public class Sense
{
  public static void motemain()
  {
    while(true)
    {
      // Turn on a led
      Leds.setLed(0,true);
      // Reads actual external temperature and writes it on serial port
      System.out.println("Temp = " + Temperature.getValue());
      // Reads actual internal temperature and writes it on serial port
      System.out.println("TempBoard = " + Temperature.getBoardValue());
      // Reads actual external pressure and writes it on serial port
      System.out.println("Pressure = " + Pressure.getValue());
      // Reads actual brightness level and writes it on serial port
      System.out.println("Light = " + Light.getValue());
      // Writes system time (in second) on serial port
      System.out.println("Time: " + VirtualSense.getSecond());

      Thread.sleep(1000);
    }
  }
}
```

Sense application senses every second the levels of all on board sensors (external temp, internal temp, pressure and brightness level) and writes the values on serial port.

9.4 Vscaling example

Frequency scaling application. Scale MCU frequency every seconds at all supported levels.

Listing 5: VScaling.java

```
// Libraries
import javax.virtualsense.powermanagement.PowerManager;
import javax.virtualsense.actuators.Leds;

public class VScaling
{
  public static void motemain()
  {
    /* Slow down the system clock (normally it is configured at 10 ms) to
       reduce power consumption leaves the CPU in the LPM3 state */
    PowerManager.setSystemClockMillis(1000);

    while(true)
    {
      // Sets MCU frequency @ 25MHZ
      PowerManager.setMCUFrequency(PowerManager.MCU_25MHZ);
      System.out.println("25MHZ: ");
      CPUBurst();
      Thread.sleep(1000);
      // Sets MCU frequency @ 20MHZ
      PowerManager.setMCUFrequency(PowerManager.MCU_20MHZ);
      System.out.println("20MHZ: ");
      CPUBurst();
      Thread.sleep(1000);
      // Sets MCU frequency @ 16MHZ
      PowerManager.setMCUFrequency(PowerManager.MCU_16MHZ);
      System.out.println("16MHZ: ");
      CPUBurst();
      Thread.sleep(1000);
      // Sets MCU frequency @ 12MHZ
      PowerManager.setMCUFrequency(PowerManager.MCU_12MHZ);
      System.out.println("12MHZ: ");
      CPUBurst();
      Thread.sleep(1000);
      // Sets MCU frequency @ 8MHZ
      PowerManager.setMCUFrequency(PowerManager.MCU_8MHZ);
      System.out.println("8MHZ: ");
```

```
        CPUBurst();
        Thread.sleep(1000);
        // Sets MCU frequency @ 4MHZ
        PowerManager.setMCUFrequency(PowerManager.MCU_4MHZ);
        System.out.println("4MHZ: ");
        CPUBurst();
        Thread.sleep(1000);
      }
    }

    public static void CPUBurst()
    {
          for(int i = 0; i < 10000; i++);
    }
  }
```

VScaling application perform the sum of 10000 integers (implemented by the function **CPUBurst**) at all supported MCU frequency level (Supported frequencies: 25MHz, 20MHz, 16MHz, 12MHz, 8MHz, 4MHz), adding a delay of one second between each change. This application is useful for estimate the power consumption at ever frequency level.

9.5 Multi Thread example

Blink example using Multiple Threads. This application is formed by 3 class:

- **MultiThreadBlink.java** Containing motemain method (main thread).
- **Th.java** Defines the behavior of all thread running concurrent of main (motemain).
- **Buffer.java** Defines an shared object between all thread.

Listing 6: MultiThreadBlink.java

```
// Libraries
import javax.virtualsense.actuators.Leds;
import javax.virtualsense.powermanagement.PowerManager;

public class MultiThreadBlink
{
  public static void motemain()
  {
    // Creates shared buffer and threads
    Buffer b = new Buffer();
    Th thread1 = new Th(b, 1000, (short)1);
    Th thread2 = new Th(b, 500, (short)2);

    // Starts threads
    thread1.start();
    thread2.start();

    while(true)
    {
      Leds.setLed(1,false);
      Leds.setLed(2,false);
      b.temp -= 4;
      Thread.sleep(2000);
      System.out.print("tmp: " + b.temp);
    }
  }
}
```

The **motemain** method creates a shared buffer and two threads that control led 1 and 2. Sleep period of two thread is half seconf for once and second for another so that the leds blinking one at double speed respect to the first. In execution time motemain, every two seconds, turn off all led and decrease share buffer of 4.

Listing 7: Th.java

```
// Libraries
import javax.virtualsense.actuators.Leds;

public class Th extends Thread
{
  Buffer myTemp;  // Shared buffer between all threads
  int myTime; // Sleep periode
  short myLed;  // Led controlled

  public Th(Buffer temp, int time, short led)
  {
    this.myTemp = temp;
    this.myTime = time;
    this.myLed = led;
  }

  public void run()
  {
    boolean state = true;

    while(true)
    {
      Leds.setLed(this.myLed, state);
      this.myTemp.temp++;
      Thread.sleep(this.myTime);
      state =! state;
      System.out.println("My led is : " + this.myLed);
    }
  }
}
```

Every thread in addition to flash the controlled led at frequency specified, increase the buffer of an unity every execution cycle.

Listing 8: Buffer.java

```
public class Buffer
{
  // Shared object is an integer modified by all threads
  public int temp;
}
```

Buffer class, composed of a single integer, is shared from all thread which change its value at all cycle. The content of buffer class is prints of serial port by **motmain** thread. Is useful control its convergence to a constant value to ensure the effective execution of all threads.

9.6 Radio Test example

Example application that use radio functionality for exchange information between a sink and more node sender. Communications is regulated by a **minpath** protocol that manage routing functionality. Application is formed by 2 class:

- **MultiThreadBlink.java** Defines the behavior of sink and sender nodes.
- **MinPathProtocol.java** Extends class Protocol to implement minpath routing algorithm.

The RadioTest class define the behavior of all node. The task of sink or sender on the network is define by a **nodeId** of mote; if a mote has **nodeId** like 1 will behave as sink otherwise as sender.

Listing 9: RadioTest.java

```
// Libraries
import javax.virtualsense.network.Network;
import javax.virtualsense.network.Packet;
import javax.virtualsense.actuators.Leds;
import javax.virtualsense.powermanagement.PowerManager;
import javax.virtualsense.VirtualSense;

public class RadioTest
{
  public static void motemain()
  {
    short nodeId = VirtualSense.getNodeId();

    if (nodeId == 1)
    { // I'am the sink init a null protocol
      Network.init();
      sink();
          }
    else
    { // I'am the sender
      Network.init(new MinPathProtocol());
      sender(nodeId);
    }
  }

  public static void sink()
  {
    System.out.println("SINK!!!");
```

```
      new Thread()
      {   // The interest sender thread
        public void run()
        {
          System.out.println("Starting interest thread!!!");
                  byte i = -126;
                  while(true)
          {
            Thread.sleep(15000);
            byte d[] = new byte[3];
            d[0] = 0; // MinPathProtocol.INTEREST;
            d[1] = 0; // num hop
            d[2] = i; // epoch
            i++;
            Packet p = new Packet(d);
            Network.send(p);
            VirtualSense.printTime();
            System.out.println(" INTEREST");
          }
        }
      }.start();
              Thread.yield();

      System.out.println("Receiver thread!!!");
              while(true)
      {
        Packet p = Network.receive();
        VirtualSense.printTime();
        System.out.println("Packet received from: " + p.getSender());
        //System.out.println(javax.virtualsense.radio.Radio.getSenderId());
        byte data[] = p.getData();

        for(int i = 0; i < data.length; i++)
        {
          Leds.setLed(1,true);
          System.out.print("-");
          System.out.print(data[i]);
          Leds.setLed(1,false);
                }
                System.out.println("");
                System.out.println("");
```

```
            }
    }

    public static void sender(short nodeId)
    {
      byte i = -127;
      boolean state = true;
      while(true)
      {
        Thread.sleep(1200);
        byte data[] = new byte[90];
        data[0] = 1; // MinPathProtocol.DATA;
        data[1] = 1; // packet should be forwarded to the sink
        data[2] = i; // this is the data
        data[3] = (byte)(nodeId>>8); // this is node id
        data[4] = (byte)(nodeId & 0xff); // this node id

        for(byte h = 5; h<data.length; h++)
          data[h] = h;
        i++;
        Leds.setLed(0, state);
        Packet p = new Packet(data);
        Network.send(p);
        VirtualSense.printTime();
        System.out.println(" -- SENDER packet sent");

        state =! state;
      }
    }
  }
```

On sink node there are two concurrent threads once that writes on serial port incoming packets from receivers nodes, and the other that defines the epochs within the network. The threads on sink node are both generated by class **RadioTest**. Also on sender nodes there are two threads once that sends packet to Sink and the others that route packets. The thread that manage packets routing is generated by class **MinPathProtocol** that sends broadcast packets to find the **minpath** between nodes.

Listing 10: MinPathProtocol.java

```
// Libraries
import javax.virtualsense.network.*;
import javax.virtualsense.actuators.Leds;
import javax.virtualsense.radio.Radio;
import javax.virtualsense.VirtualSense;

public class MinPathProtocol extends Protocol
{
  private byte minHops = (byte)127;
  private byte epoch = (byte)-127;
  short nodeId = VirtualSense.getNodeId();
  private static byte data[] = new byte [10]; // to fix a GC problem

  protected void packetHandler(Packet received)
  {
    data = received.getData();

    if(data[0]==0)
    { // INTEREST MESSAGE
      Leds.setLed(2,true);
      if(data[2] > epoch)
      { // new epoch start -- reset routing table
        epoch = data[2];
        super.bestPath = -1;
        minHops = (byte)127;
      }
      if(data[1] < this.minHops)
      {
        VirtualSense.printTime();
        System.out.println(" Routing updated ");
        this.minHops = data[1];
        super.bestPath = Radio.getSenderId();

        // in this case we need to forward the interest
        // and increment the hop counter
        data[1]+=1;
        Packet forward = new Packet(data);
        Thread.sleep(50);
        super.sendBroadcast(forward);
      }
```

```
        Leds.setLed(2,false);
      }
      if (data[0] == 1) //DATA
      {
        if(data[1] == 1)// the packet should be forwarded to the sink 1.. we
            are a router
        {
          Leds.setLed(4,true);
          VirtualSense.printTime();
          System.out.println(" Forward packet to the sink");
          Thread.sleep(50);
          super.send(received); // to paper filtering
          Leds.setLed(4,false);
        }
        else    // data is for the node. We are the receiver
        {
          Leds.setLed(5,true);
          VirtualSense.printTime();
          System.out.println(" Data is for us");
          super.notifyReceiver();
          Leds.setLed(5,false);
        }
      }

    }
  }
```

10 Create a new application

For create a new VirtualSense application, named for example **"new_app1"**, you must take the following steps. On VirtualSense SDK Directory Tree create a new folder named **"new_app1"** that will contain all file of new application.

```
~/VirtualSense_SDK.1.1.0/src/
```

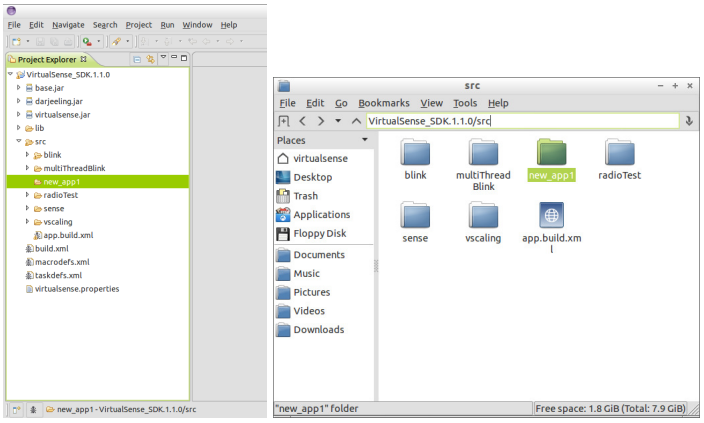


Figure 6: Create a new application folder

In the new directory named **"new_app1"** creates a new **java file** that will be contain the main method and a **xml buil file** to use for compile application. The name of java file should not respect few rules and for example can call **"NewApp1.java"** but build file must be called **"app.build.xml"**.

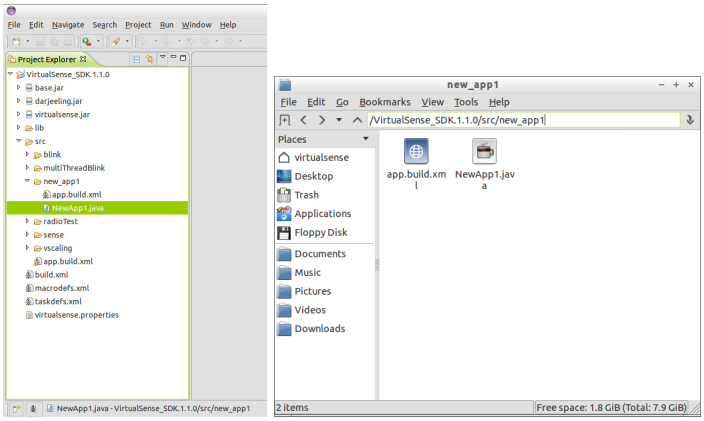


Figure 7: Create java file and build file

File **NewApp1.java**

Listing 11: NewApp1.java

```
/*
 *  NewApp1.java
 *
 *  Copyright (c) 2013 DiSBeF, University of Urbino.
 *
 *  This file is part of VirtualSense.
 *
 *  VirtualSense is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  VirtualSense is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with VirtualSense.  If not, see <http://www.gnu.org/licenses/>.
 */

/**
 * New application
 *
 * @author virtualsense
 */

import //Libraries

public class NewApp1
{
  // Definition of motemain
  public static void motemain()
  {
    while(true)
    {
      // TODO code application logic here
    }
  }
}
```

File **app.build.xml**

Listing 12: app.build.xml

```
<project basedir="../../../" default="compile">

  <!-- application directory (relative to /src/app) -->
  <property name="app.dir" value="new_app1" />

  <!-- application name and version -->
  <property name="app.name" value="new_app1" />
  <property name="app.majorversion" value="0" />
  <property name="app.minorversion" value="1" />

  <!-- the rest is included from apps.build.xml -->
  <import file="../app.build.xml" />

  <!-- includes -->
  <patternset id="app.includes">
    <include name="base.dih" />
    <include name="darjeeling.dih" />
    <include name="virtualsense.dih" />
  </patternset>

</project>
```

In file **app.build.xm** is important set properties **app.name** and **app.dir** with the name of app directory, in this case **"new_app1"**.

After created application file last step is update file **build.xml** located in:

```
~/VirtualSense_SDK.1.1.0/
```

In target **apps** add an application dir named with new directory name.

Listing 13: build.xml

```
<project name="VirtualSense_SDK">

  <import file="${basedir}/macrodefs.xml" />

  <!-- build all applications -->
  <target name="build-all-apps" depends="">
    <application dir="blink" target="hexdump" />
    <application dir="sense" target="hexdump" />
    <application dir="radioTest" target="hexdump" />
    <application dir="multiThreadBlink" target="hexdump" />
    <application dir="vscaling" target="hexdump" />
    <application dir="new_app1" target="hexdump" />
  </target>

  <!-- clean -->
  <target name="clean">
    <delete dir="${build}" />
  </target>

  <target name="run-virtualsense-blink" depends="build-all-apps">
  <distro distro="virtualsense" infusions="base, darjeeling, virtualsense"
    nativeinfusions="base, darjeeling, virtualsense" run="virtualsense"
    apps="blink"
    running="blink" target="run" />
  </target>

  <target name="run-virtualsense-multithread-blink" depends="build-all-apps">
  <distro distro="virtualsense" infusions="base, darjeeling, virtualsense"
    nativeinfusions="base, darjeeling, virtualsense" run="virtualsense"
    apps="multiThreadBlink"
    running="multiThreadBlink" target="run" />
  </target>

  <target name="run-virtualsense-radio-test" depends="build-all-apps">
  <distro distro="virtualsense" infusions="base, darjeeling, virtualsense"
    nativeinfusions="base, darjeeling, virtualsense" run="virtualsense"
    apps="radioTest"
    running="radioTest" target="run" />
  </target>

  <target name="run-virtualsense-sense" depends="build-all-apps">
  <distro distro="virtualsense" infusions="base, darjeeling, virtualsense"
    nativeinfusions="base, darjeeling, virtualsense" run="virtualsense"
    apps="sense"
    running="sense" target="run" />
  </target>

  <target name="run-virtualsense-vscaling" depends="build-all-apps">
  <distro distro="virtualsense" infusions="base, darjeeling, virtualsense"
    nativeinfusions="base, darjeeling, virtualsense" run="virtualsense"
    apps="vscaling"
    running="vscaling" target="run" />
  </target>
```

Add a new target named **"run-virtualsense-newapp1"** for compile and instal the application on motes.

```
  <target name="run-virtualsense-newapp1" depends="build-all-apps">
  <distro distro="virtualsense" infusions="base, darjeeling, virtualsense"
    nativeinfusions="base, darjeeling, virtualsense" run="virtualsense"
    apps="new_app1"
    running="new_app1" target="run" />
  </target>
</project>
```

11 Install an application

Next we will explain how to install a sample application, for example the blink sample application. There are two methods to install an application on a VirtualSense node:

- Using the **JTAG serial Interface**.
- Using the **Boot Strap Loader**.

11.1 Installation by JTAG serial interface

If you prefer using Eclipse open it and open **build.xml** situated on:

`VirtualSense/DJ_VirtualMachine.1.0/darjeeling.1.1`

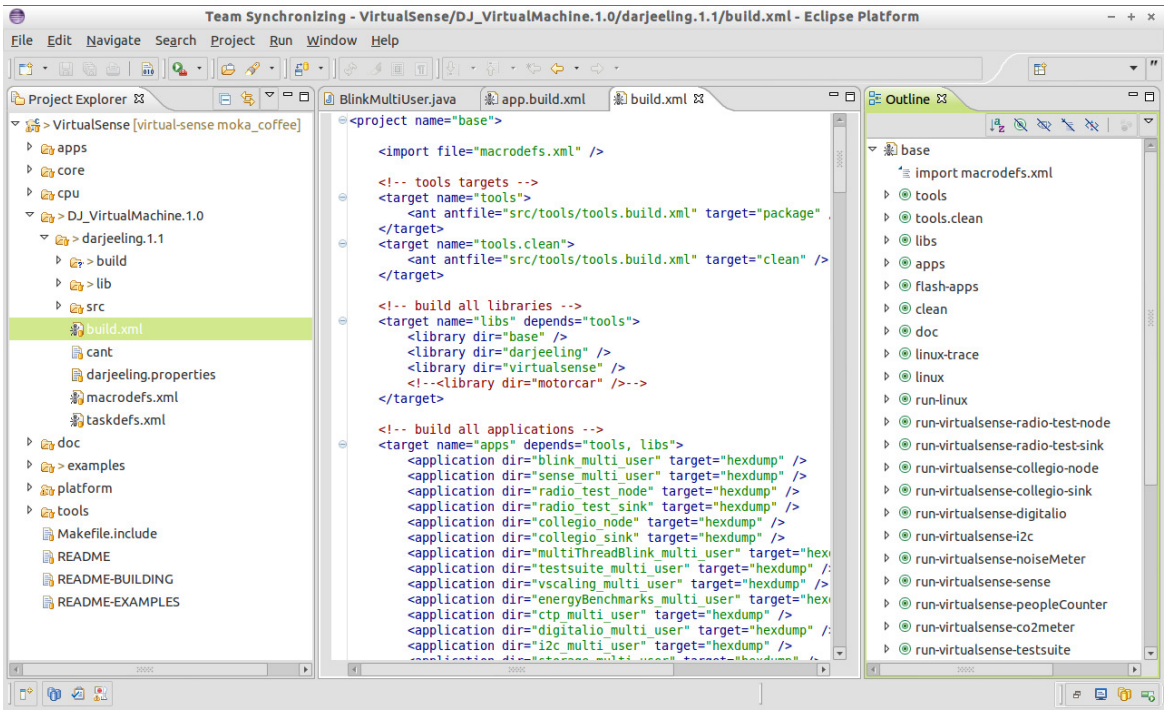


Figure 8: Installation by JTAG: file build.xml on Eclipse.

Otherwise open the same path using the Terminal.

```
virtualsense@ubuntu:~$ cd git/virtual-sense/VirtualSense/DJ_VirtualMachine
                          .1.0/darjeeling.1.1/
virtualsense@ubuntu:~/git/virtual-sense/VirtualSense/DJ_VirtualMachine.1.0/
                    darjeeling.1.1$ ls
build.xml  cant  darjeeling.properties  lib  macrodefs.xml  src  taskdefs.
    xml
virtualsense@ubuntu:~/git/virtual-sense/VirtualSense/DJ_VirtualMachine.1.0/
                    darjeeling.1.1$
```

Connect JTAG connector to VirtualSense ProgDebug layer and JTAG USB cable to PC. Make sure that it is mounted on port **ttyACM0**, using the command **dmesg**.

```
virtualsense@ubuntu:~$ dmesg
...
...
[ 1504.442315] usb 2-2.1: Product: Texas Instruments MSP430-JTAG
[ 1504.442317] usb 2-2.1: Manufacturer: Texas Instruments
[ 1504.442319] usb 2-2.1: SerialNumber: 49FF6053A7D53D23
[ 1504.445174] cdc_acm 2-2.1:1.0: This device cannot do calls on its own.
    It is not a modem.
[ 1504.445206] cdc_acm 2-2.1:1.0: ttyACM0: USB ACM device        <<<<<
virtualsense@ubuntu:~$
```

Note. If FTDI 3V3 Jumper on VirtualSense ProgDebug layer is placed you must plug into the PC even the programmer USB cable to provide power to the node.

Now to compile and install the application using Eclipse, open **Outline** tab. Search **run-virtualsense-blink** target, right click on it and select **Run As** > **1 Ant Build**.

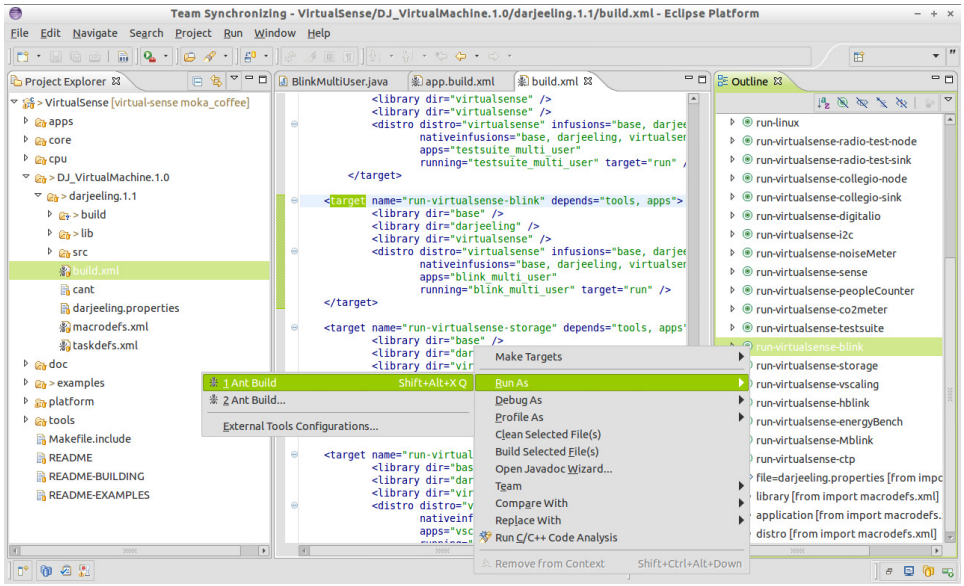


Figure 9: Installation by JTAG: compiling with ant builder on Eclipse.

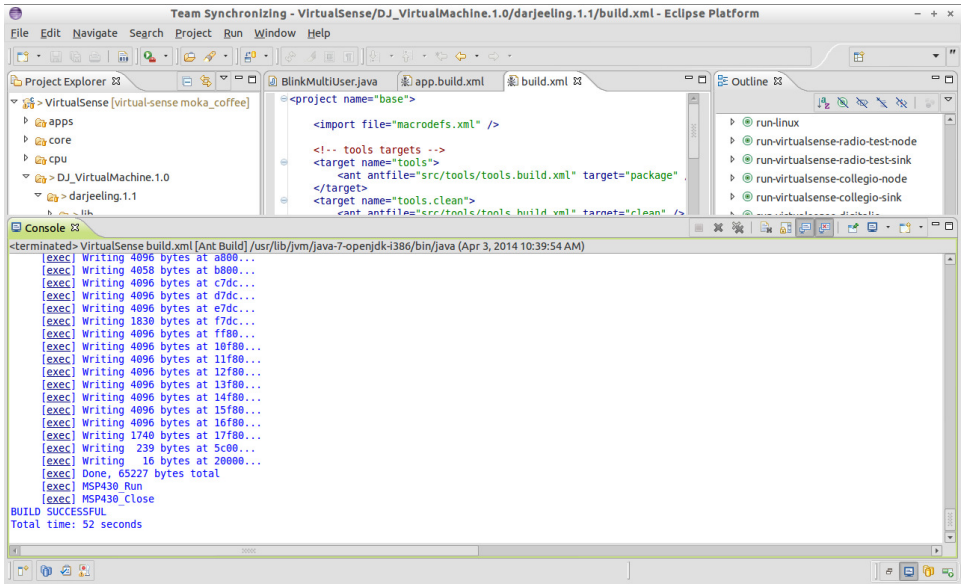


Figure 10: Installation by JTAG: compiling and installation on Eclipse done.

Instead using the Terminal run the command:

```
ant run-virtualsense-blink
```

```
virtualsense@ubuntu:~/git/virtual-sense/VirtualSense/DJ_VirtualMachine.1.0/
                      darjeeling.1.1$ ant run-virtualsense-blink
...
...
     [exec] Writing   16 bytes at 20000...
     [exec] Done, 65227 bytes total
     [exec] MSP430_Run
     [exec] MSP430_Close

BUILD SUCCESSFUL
Total time: 23 seconds
```

The application is installed and you only have to reset the node, to start the execution. You can debug the app using a serial client like **Cutecom**.

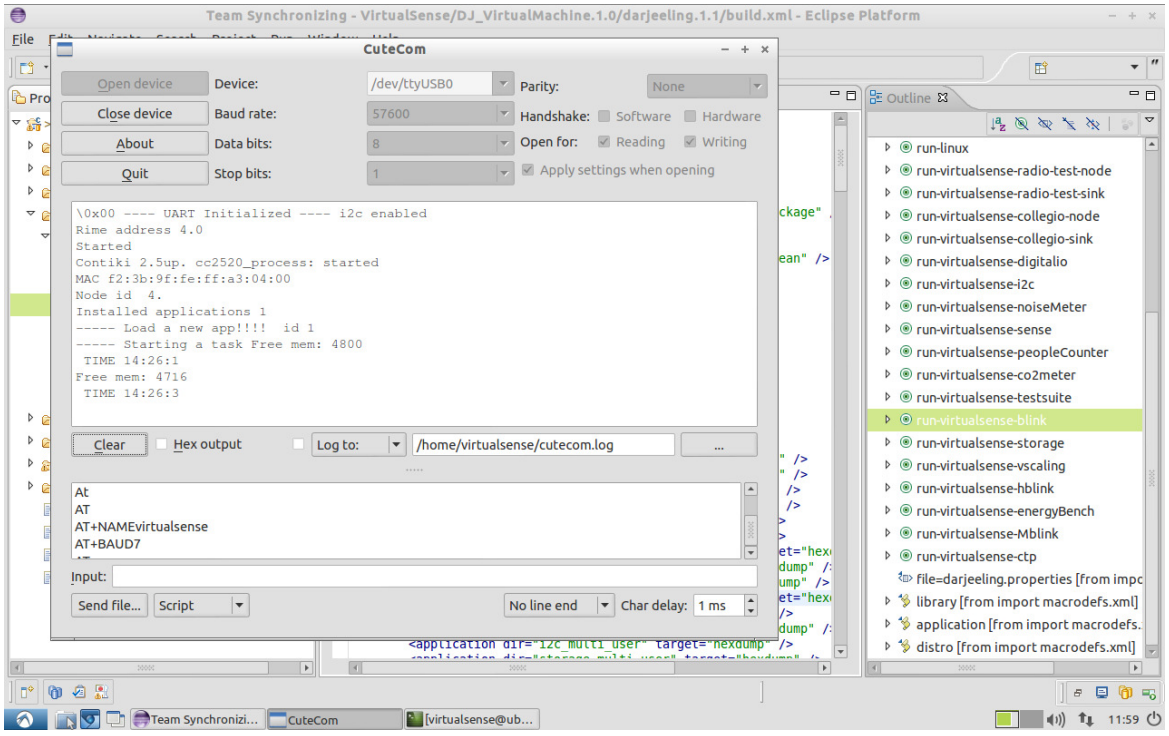


Figure 11: Installation by JTAG: app installed, cutecom serial debug.

11.2 Installation by Boot Strap Loader (BSL)

To install an app with BSL Programmer you must modify an project configuration file named **vm.contiki.msp430.build.xml** located in:

```
VirtualSense/DJ_VirtualMachine.1.0/darjeeling.1.1/src/vm
```

Once opened the file, at the end of it, you can found the instruction used for install the app on the node.

Listing 14: vm.contiki.msp430.build.xml

```
    <target name="run" depends="compile, applicationflasher">

      <property name="msp430-port" value="/dev/ttyACM0" />

       <!-- this is the programming command under Linux OS using mspdebug
           -->
      <exec executable="mspdebug">
        <arg line="tilib" />
        <arg line="-j" />
        <arg line="-d ${msp430-port}" />
        <arg line="'prog  ${vm.executables}/${distro}/virtualsense.txt'" />
      </exec>
      <!-- END of programming command under Linux OS using mspdebug -->
    </target>

</project>
```

To use BSL Programmer change it as following:

Listing 15: vm.contiki.msp430.build.xml

```
    <target name="run" depends="compile, applicationflasher">

      <property name="msp430-port" value="/dev/ttyUSB0" />

       <!-- this is the programming command under Linux OS using BSL
          Programmer -->
      <exec executable="virtualsense-bsl">
        <arg line="-d ${msp430-port}" />
        <arg line="flash-bsl5" />
        <arg line="--long-password" />
        <arg line="'prog  ${vm.executables}/${distro}/virtualsense.txt'" />
      </exec>
      <!-- END of programming command under Linux OS using BSL Programmer
         -->
    </target>

</project>
```

Now if you prefer using Eclipse open it and open **build.xml** situated on:

```
VirtualSense/DJ_VirtualMachine.1.0/darjeeling.1.1
```

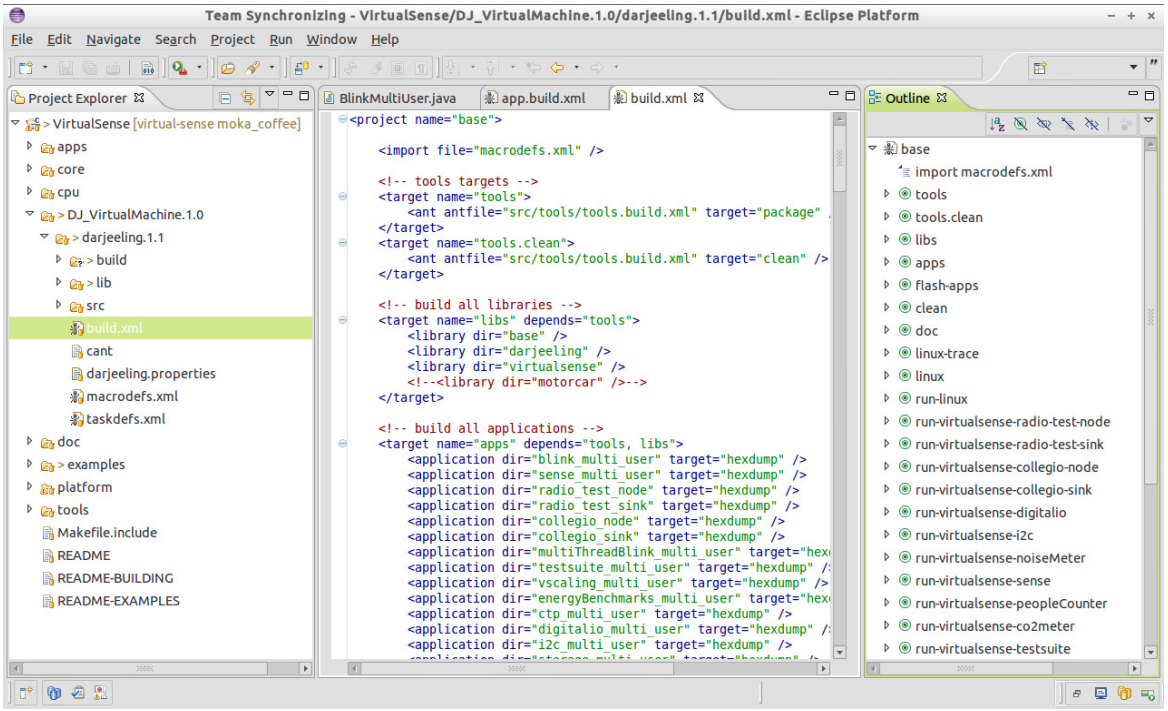


Figure 12: Installation by BSL: file build.xml on Eclipse.

Otherwise open the same path using the Terminal.

```
virtualsense@ubuntu:~$ cd git/virtual-sense/VirtualSense/DJ_VirtualMachine
                          .1.0/darjeeling.1.1/
virtualsense@ubuntu:~/git/virtual-sense/VirtualSense/DJ_VirtualMachine.1.0/
                     darjeeling.1.1$ ls
build.xml  cant  darjeeling.properties  lib  macrodefs.xml  src  taskdefs.
    xml
virtualsense@ubuntu:~/git/virtual-sense/VirtualSense/DJ_VirtualMachine.1.0/
                     darjeeling.1.1$
```

Connect the USB cable of the VirtualSense ProgDebug and check, using the **dmesg** command, that USB serial interface is connected on **ttyUSB0**.

```
virtualsense@ubuntu:~$ dmesg
...
...
[ 2324.076168] usb 2-2.2: Product: FT232R USB UART
[ 2324.076176] usb 2-2.2: Manufacturer: FTDI
[ 2324.076182] usb 2-2.2: SerialNumber: AE01908B
[ 2324.083090] ftdi_sio 2-2.2:1.0: FTDI USB Serial Device converter
   detected
[ 2324.083233] usb 2-2.2: Detected FT232RL
[ 2324.083240] usb 2-2.2: Number of endpoints 2
[ 2324.083247] usb 2-2.2: Endpoint 1 MaxPacketSize 64
[ 2324.083253] usb 2-2.2: Endpoint 2 MaxPacketSize 64
[ 2324.083259] usb 2-2.2: Setting MaxPacketSize 64
[ 2324.085737] usb 2-2.2: FTDI USB Serial Device converter now attached to
   ttyUSB0   <<<<<
virtualsense@ubuntu:~$
```

At this time to compile and install the application using Eclipse, open **Outline** tab. Search **run-virtualsense-blink** target, right click on it and select **Run As** > **1 Ant Build**.

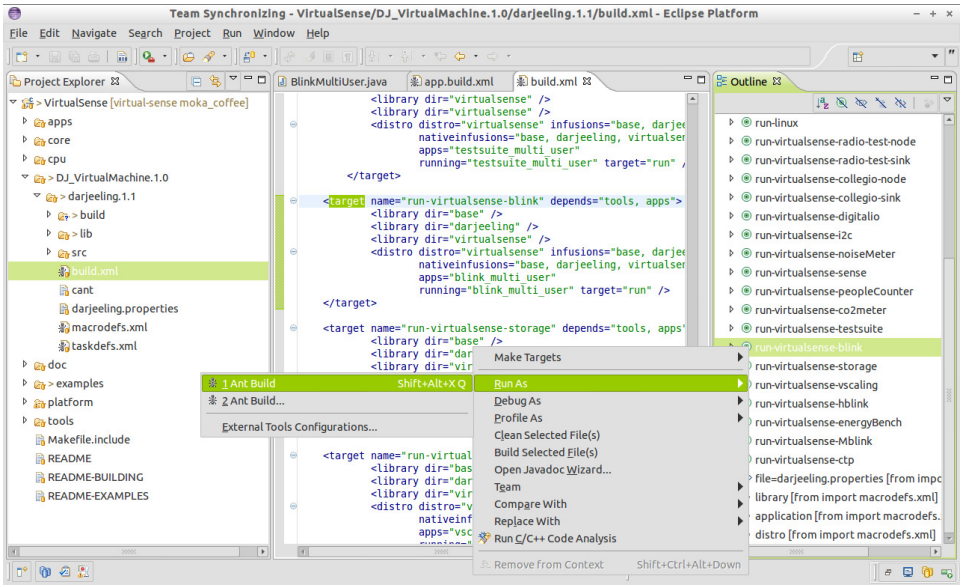


Figure 13: Installation by BSL: compiling with ant builder on Eclipse.

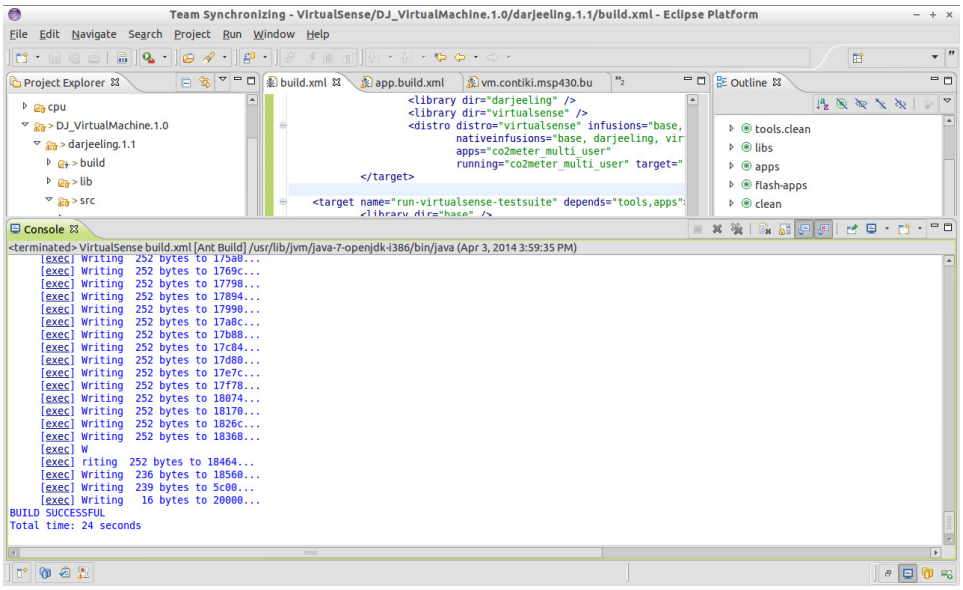


Figure 14: Installation by BSL: compiling and installation on Eclipse done.

Instead using the Terminal run the command:

```
ant run-virtualsense-blink
```

```
virtualsense@ubuntu:~/git/virtual-sense/VirtualSense/DJ_VirtualMachine.1.0/
                      darjeeling.1.1$ ant run-virtualsense-blink
...
...
     [exec] Writing  239 bytes to 5c00...
     [exec] Writing   16 bytes to 20000...

BUILD SUCCESSFUL
Total time: 23 seconds
```

Now the application is installed. You can debug the app using a serial client like **Cutecom**.

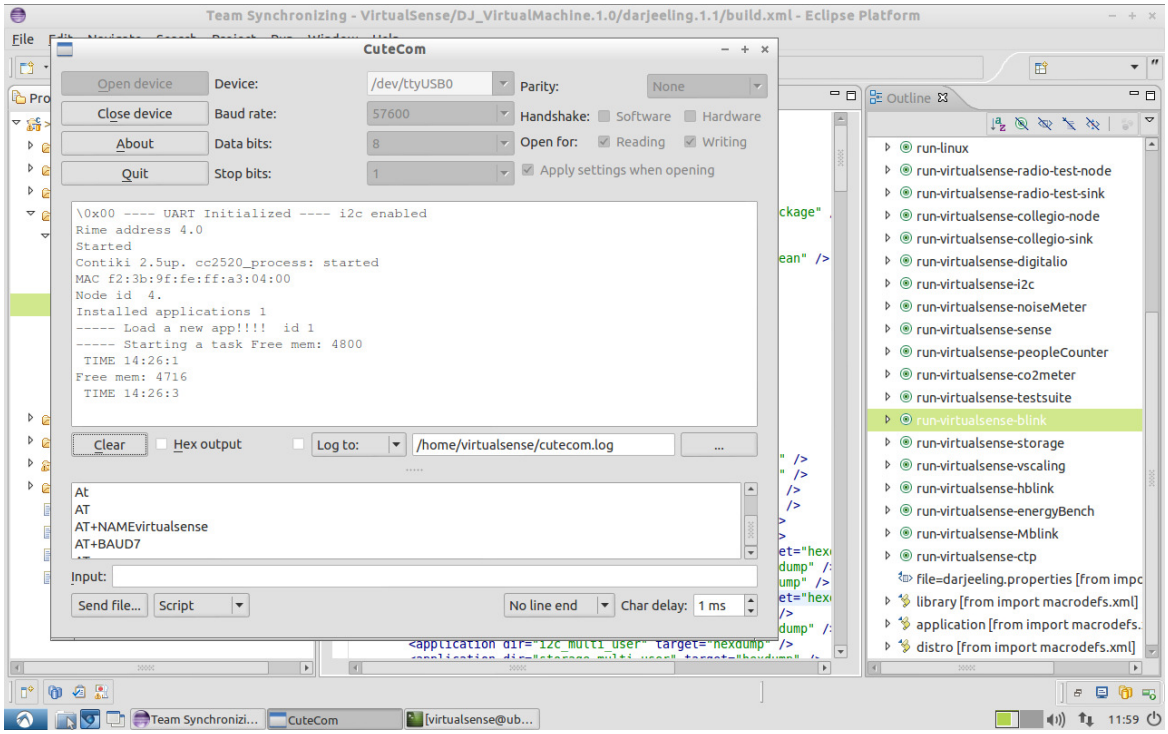


Figure 15: Installation by BSL: app installed, cutecom serial debug.