Internship at Sentilla

Chris Merlin
March 08 - June 08
Overview

- The Sentilla Co. and The Product
- The smallest platform to carry a JVM
- Particularities of Coding in Java
- Application Engineering
Sentilla

• Formerly Moteiv (2003)
  o Moteiv was focused on hardware (products: tmote-sky, tmote-invent, etc.)
  o Moteiv used TinyOS
• Sentilla (2007) now oriented toward software
  o Adopted Java
    ▪ Simple programming language
    ▪ 6 million Java programmers in the world
• Small team
• Venture-backed Company: Onset, Claremont Creek
The Sentilla Product: Sentilla Work

• Focus of Sentilla: to provide Software for quick *Pervasive Computing* deployments using Java

• IDE (Sentilla Work):
  ○ Lets programmers write Java Code
  ○ Comes with help and tutorials
  ○ Allows to burn motes with firmware
  ○ Has to be understood as Windows of WSN:
    ▪ Provides APIs, Drivers, libraries
    ▪ Client can run Java applets to interface with Internet or other elements of computer
    ▪ Manages applications
    ▪ Users write applications for their needs
Firmware / MAC

• MAC:
  ○ Part of firmware: cannot be changed
  ○ MAC is Low-Power-Listening scheme: value of LPL interval fixed, but can be chosen to be one of 3 values
• Passphrase for data encryption: slows communications by factor of 3-4
• Communication channel fixed

makes subnetwork
Overview

- The Sentilla Co. and The Product
- The smallest platform to carry a JVM
- Particularities of Coding in Java
- Application Engineering
The JVM

• Core based on Ti MSP430
  - 16bit, 8MHz CPU
  - 48K flash on CPU
  - 10K RAM
  - 1MB external flash

• Runs a Java Virtual Machine

• Now CLDC 1.0 compliant
Advantages of Java

• Object-Oriented programming language:
  o Particularly well adapted to wireless sensor nodes
  o Allows to instantiate different objects depending on the mote ID

```java
public static void motemain()
{
    long id = [...];
    if (id == mote1_id)
        new app1();
    else
        new app2();
}
```

now, code for sender and receiver can be clearly separated
import com.sentilla.io.*;
import com.sentilla.system.*;
import javax.measure.*;
public class SensorApp {  
  /**
   * Compact programming language
   */
  public static class SensorMessage implements Serializable {
    public long address;
    public Measurable<Temperature> value;
  }
  public static void motemain() {
    Sensor<Temperature> temp = SensorDriver.create("mcutemp".Temperature.class);
    Sensor sender = SensorDriver.create("collect");
    SensorMessage msg = new SensorMessage();
    msg.address = PropertyDriver.open("mac41").readLong();
    while (true) {  
      msg.value = temp.read();
      if (msg.value.doubleValue(SI.CELSIUS) > (double)50) {
        sender.send(msg);
      }
    }
  }
  try { Thread.sleep(1000); } catch (Exception e) { e.printStackTrace() }
}

public class MultihopOscilloscopeC implements Serializable {
  public long address;
  public Measurable<Temperature> value;
  /**
   * Compact programming language
   */
  public static class SensorMessage implements Serializable {
    public long address;
    public Measurable<Temperature> value;
  }
  public static void motemain() {
    Sensor<Temperature> temp = SensorDriver.create("mcutemp".Temperature.class);
    Sensor sender = SensorDriver.create("collect");
    SensorMessage msg = new SensorMessage();
    msg.address = PropertyDriver.open("mac41").readLong();
    while (true) {  
      msg.value = temp.read();
      if (msg.value.doubleValue(SI.CELSIUS) > (double)50) {
        sender.send(msg);
      }
    }
  }
  try { Thread.sleep(1000); } catch (Exception e) { e.printStackTrace() }
}

public class MultihopOscilloscopeC implements Serializable {
  public long address;
  public Measurable<Temperature> value;
  /**
   * Compact programming language
   */
  public static class SensorMessage implements Serializable {
    public long address;
    public Measurable<Temperature> value;
  }
  public static void motemain() {
    Sensor<Temperature> temp = SensorDriver.create("mcutemp".Temperature.class);
    Sensor sender = SensorDriver.create("collect");
    SensorMessage msg = new SensorMessage();
    msg.address = PropertyDriver.open("mac41").readLong();
    while (true) {  
      msg.value = temp.read();
      if (msg.value.doubleValue(SI.CELSIUS) > (double)50) {
        sender.send(msg);
      }
    }
  }
  try { Thread.sleep(1000); } catch (Exception e) { e.printStackTrace() }
}
Advantages of Java

• Dynamic memory allocation
  o Garbage collector
  o Supported Objects:
    ▪ Strings
    ▪ Arrays of variable size
• Native code still used for interfacing with hardware
• Multi-threading

• 6,000,000+ Java Programmers in the World
  (200,000 TinyOS)
Disadvantages of Java

- Slower than native code (4 orders of magnitude)
  - I measured 17ms to toggle LEDs!
- JVM takes a large part of the available resources
  - Forces to use swapping: slow
Overview

• The Sentilla Co. and The Product
• The smallest platform to carry a JVM
• Particularities of Coding in Java
• Application Engineering
Driver Instantiation

- Since OO programming language, every layer of the stack becomes an Object
- To send a packet: instantiate a SenderDriver as such:

  ```java
  public class Message implements Serializable {
    public float data;
  }
  ...
  Sender sender = SenderDriver.create("local");
  Message msg = new Message();
  sender.send(msg);
  ```

- Instantiate Receiver, Sender, Leds, messages, DigitalPins, VoltageAdc, etc.
Interrupts

• Java offers a relatively simple way to handle events/interrupts
• Need to instantiate an InterruptPin (for instance)
• Two solutions:
  ○ Invoke blocking calls: the thread hangs until specified timeout or event completed
    ▪ `pin.block().wait(timeout);`
    ▪ Blocking call stops thread ➔ may need to create other threads for rest of program
  ○ If loop cannot be stopped, keep checking whether event has happened or not
    ▪ `pin.isDone();`
Interrupts

- Receiver has a blocking call: can specify timeout after which thread continues if no packet received (ACKs)
- InterruptPin’s can specify rising / falling edge, or either
- Go hand-in-hand with atomicity concerns
Atomicity

- Handled very differently than in TinyOS x.x
  - No tasks
  - No synchronous or asynchronous events
  - No events
- Use of lock (a mutex):
  - Declare an Object that will be your mutex
  - Enter a Synchronized block; locks mutex
  - The code within Synchronized blocks over the same mutex is mutually exclusive
  - At end of block, release mutex (convention)
public class LeftMonitorThread extends Thread{
    public LeftMonitorThread(){
        msg.countIn = 0;
        msg.countOut = 0;
    }

    public void run(){
        int direction;
        while(true){
            direction = 0;
            leftBeam.edge(false, true);
            synchronized(lock){
                lastTimeLeftTripped = java.lang.System.currentTimeMillis();
                lock.notifyAll();
            }
            Thread.sleep(100);
            if (!leftBeam.read()){  
                synchronized(lock){
                    lastTimeLeftTripped = 0;
                    direction = -1;
                }
                lock.notifyAll();
            } else{
                synchronized(lock){
                    if (lastTimeLeftTripped -
                        lastTimeRightTripped < 700){
                        msg.countOut++;
                        lastTimeLeftTripped = 0;
                        direction = -1;
                    }
                    lock.notifyAll();
                }
            }
            displayLeds(direction, (short)2,
                        (direction == 0 ? false : true));
        }
    }
}

If other beam broken less than 700ms ago: person

else{
    synchronized(lock){
        if (lastTimeLeftTripped -
            lastTimeRightTripped < 700){
            msg.countOut++;
            lastTimeLeftTripped = 0;
            direction = -1;
        }
        lock.notifyAll();
    }
    displayLeds(direction, (short)2,
                (direction == 0 ? false : true));
}

Probably was nothing (a fly)
Tricks and Customs

• Coding in Java for extremely small platforms unusual
• Same ideas apply as when first cell phones appeared and were running Java

• Do not keep re-instantiating / Reuse objects
  ○ Take allocations out of loop
• Have as many blocking commands as possible
  ○ Avoid loop that keeps checking status (keeps CPU in sleep mode)
• Threads are expensive (2)
• Keep number of classes low to reduce look-up
• Use fewer LEDs (3mA per LED) / blink them
• Avoid frequent packets
• Use raw values whenever possible (for accelerometer, etc.)
Debugging

- Java throws Exceptions when normal flow of execution is interrupted
- Exceptions are propagated and appear on console
- The JCreate has 8LEDs

- Problem: native code cannot be debugged without help of JTag
Overview

- The Sentilla Co. and The Product
- The smallest platform to carry a JVM
- Particularities of Coding in Java
- Application Engineering
Application Engineering

- Consists in a team of 3 engineers
- The team is the service side of the business model
  - Writing applications for the office to showcase
  - Developing applications for conferences
  - Providing services for customer’s specific needs (energy scavenging, deployment challenges, etc.)
- Developed JCreate as a prototyping platform:
  - Has a MSP430 based core
  - Has an accelerometer
  - 8 LEDs
  - 2 AAA batteries
The JCreate Accelerometer

- 3-axis accelerometer built in JCreate
- Formerly, only 1 channel of ADC could be sampled before return statement
- Code for multiple channel samples:
  - Sequential mode
  - Channels to sample placed in array of variable size (in TinyOS!)
  - Values returned in buffer of variable size

➤ For 3 read operations, time went from 30ms to 10ms
Java Accelerometer Driver

- Instantiates ADC pins (constructor argument is pin number), allocates memory
- Starts the Accelerometer chip
- Sets sensitivity
- Keeps track of monitored axis
- Gets raw values (4B int)
  - $x$ is raw value
  - $\text{Voltage} = x >> 20$ (or division by $1024^2$)
Applications: POV

- Persistence of Vision
- Display a sentence onto mote’s LEDs by moving JCreate back and forth
- Accelerometer starts display sequence according to hand position
• Count people in and out of rooms
• Emitter:
  o 2 IR beams sent by emitter (9V individual power)
  o 36KHz and 56kHz to discriminate between beams
  o Brought power from 3.5W to 1.75W
• Receiver:
  o Mote powers and reads two IR receivers
  o Depending on which beam cut first: direction of person walking known
  o Uses interrupt pins: CPU waits for interrupt (sleeps) => low energy use
• Monitor humidity, light and temperature
• **Use of** `collect protocol`
  ○ Many-to-one
  ○ **Routing over several hops**
  ○ Every node sends its data
  ○ Buggy protocol
• **Information reported to website:** data displayed over time
Questions and Answers