# INDEX

1. Introduction .................................................................................................................................................. 6
   1.1. Concepts ................................................................................................................................................ 6
   1.2. Licences and Conditions of Use ........................................................................................................... 6
   1.3. Waspmote VS Waspmote Mote Runner ................................................................................................ 7

2. Waspmote Mote Runner Dev Kit .................................................................................................................. 8
   2.1. Box content .......................................................................................................................................... 8
      2.1.1. Waspmote Mote Runner 6LoWPAN Networking Kit ................................................................. 8
      2.1.2. Waspmote Mote Runner 6LoWPAN Lab Kit .............................................................................. 9
   2.2. General and safety information ........................................................................................................... 10
   2.3. Assembly .............................................................................................................................................. 12
   2.4. Powering the Nodes .............................................................................................................................. 17
      2.4.1. Battery ........................................................................................................................................ 17
      2.4.2. USB ............................................................................................................................................. 18
      2.4.3. Solar Panel .................................................................................................................................. 19

3. Waspmote’s Hardware ................................................................................................................................... 21
   3.1. Modular Architecture .......................................................................................................................... 21
   3.2. Specifications ...................................................................................................................................... 21
   3.3. Block Diagram ................................................................................................................................... 22
   3.4. Electrical Data ................................................................................................................................... 23
   3.5. I/O ....................................................................................................................................................... 24
      3.5.1. Analog ........................................................................................................................................ 25
      3.5.2. Digital ........................................................................................................................................ 25
      3.5.3. UART ........................................................................................................................................ 26
      3.5.4. I2C .............................................................................................................................................. 26
      3.5.5. SPI ............................................................................................................................................. 26
      3.5.6. USB ............................................................................................................................................ 27
   3.6. Real Time Clock-RTC .......................................................................................................................... 27
   3.7. LEDs ................................................................................................................................................... 28
   3.8. Communication Modules .................................................................................................................... 29
      3.8.1. 6LoWPAN Radios ......................................................................................................................... 29
      3.8.2. Ethernet Module ......................................................................................................................... 29

4. Architecture and System .............................................................................................................................. 30
   4.1. Concepts ............................................................................................................................................. 30
   4.2. Programming modes ............................................................................................................................. 31
      4.2.1. Reactive Programming (Asynchronous) ......................................................................................... 31
      4.2.2. Imperative programming (Synchronous) .................................................................................... 32
6.1.3. Linux / Linux (64-bit) ................................................................. 58
  6.1.3.1. Quick Start ............................................................ 58
  6.1.3.2. PATH and LD_LIBRARY_PATH Environment Variable .......... 58
  6.1.3.3. Optional: C# development .......................................... 58
6.1.4. MAC OSX ............................................................................ 58
  6.1.4.1. Quick Start ............................................................ 58
  6.1.4.2. PATH Environment Variable ........................................ 59
  6.1.4.3. Optional: C# development .......................................... 59
6.1.5. Optional: Eclipse Integration .................................................. 59
6.2. Firmware Installation for Waspmote ............................................. 59
  6.2.1. Flashing the hex image .................................................... 59
    6.2.1.1. Loading the firmware using avrdude ............................. 61
    6.2.1.2. Loading the firmware using Atmel Studio ....................... 61
6.3. Application development ............................................................. 62
  6.3.1. Compiling applications ................................................... 62
  6.3.2. Major.Minor Version Numbers ......................................... 63
  6.3.3. Uploading applications ................................................... 63
    6.3.3.1. Uploading assemblies to simulated motes ...................... 63
    6.3.3.2. Uploading assemblies to real motes ............................ 64
  6.3.4. Debugging applications .................................................. 65
  6.3.5. Simulating applications .................................................. 66
    6.3.5.1. Timeline ............................................................ 66
  6.3.6. Managing Applications MOMA ......................................... 67
6.4. Libraries .................................................................................. 67
  6.4.1. Saguaro System ............................................................ 67
  6.4.2. Waspmote System ........................................................ 68
  6.4.3. Logger ............................................................................ 68
  6.4.4. WLIIP Gateway ............................................................. 69
  6.4.5. 6LoWPAN / MRv6 ........................................................ 69
6.5. Examples and demos ................................................................. 70
6.6. Mote Runner API Documentation .............................................. 71

7. 6LoWPAN / MRv6 ........................................................................ 72
  7.1. Introduction .......................................................................... 72
  7.2. 6LoWPAN Libraries ............................................................. 72
  7.3. Using MRv6 Libraries .......................................................... 73
    7.3.1. Setup ............................................................................ 73
    7.3.2. Running the example ................................................... 74
  7.4. Quick start development with the Waspmote Mote Runner Lab Kit .. 75
    7.4.1. Application code ......................................................... 76
    7.4.2. Retrieving values from the WSN ................................... 77

8. Support ....................................................................................... 78

9. Documentation Changelog ........................................................... 79
10. Certifications ........................................................................................................................................... 80
    10.1. CE .................................................................................................................................................. 80
    10.2. FCC ............................................................................................................................................ 81
    10.3. IC ................................................................................................................................................ 82
    10.4. Use of equipment characteristics ............................................................................................. 82
    10.5. Limitations of use ....................................................................................................................... 82

11. Maintenance ......................................................................................................................................... 84

12. Disposal and recycling ..................................................................................................................... 85
1. Introduction

1.1. Concepts

IBM and Libelium have joined efforts to offer a unique IPv6 development platform for sensor networks and the Internet of Things (IoT). By integrating the IBM Mote Runner SDK on top of Libelium Waspmote sensor platform we get a unique and powerful tool for developers and researchers interested in 6LoWPAN / IPv6 connectivity for the Internet of Things.

6LoWPAN is the acronym of IPv6 over Low power Wireless Personal Area Network. This protocol offers encapsulation and header compression mechanisms that allow IPv6 packets to be sent to and received from over IEEE 802.15.4 based networks.

The Waspmote Mote Runner Development Platform allows Developers and Researchers to study, analyse and modify the 6LoWPAN protocol in order to improve it and test new routing algorithms on top.

1.2. Licences and Conditions of Use

- The Waspmote Mote Runner Kits consist of LIBELIUM Waspmote sensor nodes modified and ready to work ONLY with the IBM Mote Runner environment. If you want to use the Waspmote IDE or other modules specific to the original Waspmote platform, you should buy one of these kits. [http://www.libelium.com/products/waspmote-mote-runner-6lowpan/#buy](http://www.libelium.com/products/waspmote-mote-runner-6lowpan/#buy)
- Although quite similar, Waspmote nodes and Waspmote Mote Runner nodes are not the same Hardware. For this reason Developers cannot interchange code, libraries or radio modules between both platforms.
- The Mote Runner SDK must be downloaded from the IBM website; it does not come with the Waspmote Mote Runner Kit. Mote Runner is distributed by IBM under an evaluation license which allows to be used at no cost for non-commercial purposes. It is highly recommended to read the license at the IBM Mote Runner Website before purchasing the kit. [http://www.zurich.ibm.com/moterunner/license.html](http://www.zurich.ibm.com/moterunner/license.html)
- Waspmote nodes included in the Kit come without any firmware installed on them. Users have to compile it using the Mote Runner SDK.
- The platform is intended to be used at this first stage by Researchers and Developers who want to get in touch with 6LoWPAN connectivity. This is perfect for laboratory study, test beds and small deployments.
- The platform is not recommended for large scale deployments, commercial usages, or final product commercialization.
- Libelium has made available a thread in the FORUM to treat any issue related to the Waspmote Mote Runner Development Platform. However, Libelium will be responsible of answering ONLY the Hardware issues. Any Software, Networking or Data related issue will be answered by the IBM developers in this thread. [http://www.libelium.com/forum/viewforum.php?f=34](http://www.libelium.com/forum/viewforum.php?f=34)
### 1.3. Waspmote VS Waspmote Mote Runner

<table>
<thead>
<tr>
<th></th>
<th>Waspmote</th>
<th>Waspmote Mote Runner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wireless Modules</td>
<td>XBee (802.15.4 2.4GHz, DigiMesh 2.4GHz, 868MHz, 900MHz, ZigBee), GPRS, 3G, RFID, WiFi Bluetooth</td>
<td>6LoWPAN radio modules (2.4 GHz and 868 MHz)</td>
</tr>
<tr>
<td>Sensor Boards</td>
<td>Yes</td>
<td>Yes, but Videocamera Board</td>
</tr>
<tr>
<td>OTA Programming</td>
<td>Yes, by Libelium (available on all XBees, GPRS, 3G and WiFi)</td>
<td>Yes, by IBM (available on the 6LoWPAN module)</td>
</tr>
<tr>
<td>SD Card</td>
<td>Yes (2GB)</td>
<td>No</td>
</tr>
<tr>
<td>IDE/SDK</td>
<td>Waspmote IDE (Open Source License)</td>
<td>Mote Runner (Evaluation License)</td>
</tr>
<tr>
<td>Programming language</td>
<td>Similar to C/C++</td>
<td>Java or C#</td>
</tr>
<tr>
<td>Sleep modes</td>
<td>Yes (three modes: sleep, deep sleep, hibernate)</td>
<td>Yes, but automated, controlled by the OS</td>
</tr>
<tr>
<td>Float variables</td>
<td>Yes. Native, easy handling</td>
<td>Only simulated handling</td>
</tr>
<tr>
<td>String variables</td>
<td>Yes. Native, easy handling</td>
<td>No, only byte arrays</td>
</tr>
<tr>
<td>Software Simulation</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Code debugging</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Programming type</td>
<td>Sequential</td>
<td>Reactive</td>
</tr>
<tr>
<td>Gateway</td>
<td>Two Options: Waspmote GW and Meshlium</td>
<td>Waspmote Mote Runner Gateway + PC for IPv4/IPv6 tunneling</td>
</tr>
<tr>
<td>Meshlium</td>
<td>Yes, Meshlium has modules to interoperate with Waspmote (XBee, 3G, WiFi, Bluetooth). Remote access, mesh networks, Sensor Parser, MySQL Database, ..</td>
<td>No, Meshlium cannot interoperate with Mote Runner</td>
</tr>
<tr>
<td>Enclosure</td>
<td>Yes (Plug &amp; Sense! line)</td>
<td>No</td>
</tr>
<tr>
<td>Bootloader</td>
<td>Yes, preinstalled and ready to work</td>
<td>No. User must install it using the Mote Runner SDK. An AVR Programmer hardware is needed. (included in the kit)</td>
</tr>
<tr>
<td>Support, Technical Assistance</td>
<td>HW + SW + Networking by Libelium</td>
<td>HW by Libelium. SW + Networking by IBM.</td>
</tr>
<tr>
<td>Recommended for experimentation with 6LoWPAN</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Recommended for industrial / commercial projects</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Recommended for new users</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Suitable for users with poor programming skills</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Recommended for experts</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time to market</td>
<td>Small</td>
<td>Medium, experimental platform, not market-focused “as is”</td>
</tr>
</tbody>
</table>


2. **Waspmote Mote Runner Dev Kit**

2.1. **Box content**

2.1.1. **Waspmote Mote Runner 6LoWPAN Networking Kit**

- 6 x Waspmote
- 6 x 6600 mAh Battery
- 6 x 6LoWPAN Module (2.4GHz or 868MHz)
- 6 x Antenna (2dB for 2.4GHz version and 0dB for 868MHz version)
- 1 x Expansion Board
- 1 x Ethernet Module
- 1 x Ethernet Cable
- 1 x Atmel AVR Programmer (with USB cable)
- 6 x Mini-USB Cable
- 6 x USB-220V Adapter

*Figure: Waspmote Mote Runner 6LoWPAN Networking kit*
2.1.2. Waspmote Mote Runner 6LoWPAN Lab Kit

- 6 x Waspmote
- 6 x 6600mAh Battery
- 6 x 6LoWPAN Module (2.4GHz or 868MHz)
- 6 x Antenna (5dB for 2.4GHz version and 4.5dB for 868MHz version)
- 6 x Mini-USB Cable
- 6 x USB-220V Adapter
- 1 x Expansion Board
- 1 x Ethernet Module
- 1 x Ethernet Cable
- 1 x Atmel AVR Programmer (with USB cable)
- 1 x Gases Board
- 1 x O₂ Sensor
- 1 x Humidity Sensor
- 1 x CO Sensor
- 1 x CO₂ Sensor
- 1 x Events Board
- 1 x Temperature Sensor
- 1 x Presence Sensor (PIR)
- 1 x Luminosity LDR Sensor
- 1 x Smart Cities Board
- 1 x Dust Sensor – PM – 10
- 1 x Smart Metering Board
- 1 x Ultrasound Sensor 0-100A
- 1 x Agriculture PRO Board
- 1 x Soil Temperature Sensor
- 1 x Weather Station WS-3000 (Anemometer + Wind + Pluviometer)
- 1 x Prototyping Board
- 1 x GPS Module
- 1 x Rigid Solar Panel 7V - 500mA
- 1 x Flexible Solar Panel 7.2V – 100mA
2.2. General and safety information

Important:

In this section, the term "Waspmote" encompasses both the Waspmote device itself and its modules and sensor boards.

- Please read carefully through the document “General Conditions of Libelium Sale and Use”.
- Do not let the electronic parts come into contact with any steel elements, to avoid injuries and burns.
- NEVER submerge the device in any liquid.
- Keep the device in a dry place and away from any liquids that might spill.
- Waspmote contains electronic components that are highly sensitive and can be accessed from outside; handle the device with great care and avoid hitting or scratching any of the surfaces.
- Check the product specifications section for the maximum allowed power voltage and amperage range and always use current transformers and batteries that work within that range. Libelium will not be responsible for any malfunctions caused by using the device with any batteries, power supplies or chargers other than those supplied by Libelium.
- Keep the device within the range of temperatures stated in the specifications section.
- Do not connect or power the device with damaged cables or batteries.
- Place the device in a location that can only be accessed by maintenance operatives (restricted area).
- In any case, keep children away from the device at all times.
- If there is an electrical failure, disconnect the main switch immediately and disconnect the battery or any other power supply that is being used.
- If using a car lighter as a power supply, be sure to respect the voltage and current levels specified in the “Power Supplies” section.
- When using a battery as the power supply, whether in combination with a solar panel or not, be sure to use the voltage and current levels specified in the “Power supplies” section.
- If a software or hardware failure occurs, consult the Libelium Web Development section.
- Check that the frequencies and power levels of the radio communication modules and the integrated antennas are appropriate for the location in which you intend to use the device.
- The Waspmote device should be mounted in a protective enclosure, to protect it from environmental conditions such as light, dust, humidity or sudden changes in temperature. The board should not be definitively installed “as is”, because the electronic components would be left exposed to the open-air and could become damaged. For a ready-to-install product, we advise our Plug & Sense! line.

General:

- Read the “General and Safety Information” section carefully and keep the manual for future reference.
- Read carefully the “General Conditions of Sale and Use of Libelium”. This document can be found at: http://www.libelium.com/development/waspmote/technical_service. As specified in the Warranty document, the client has 7 days from the day the order is received to detect any failure and report that to Libelium. Any other failure reported after these 7 days may not be considered under warranty.
- Use Waspmote in accordance with the electrical specifications and in the environments described in the “Electrical Data” section of this manual.
- Waspmote and its components and modules are supplied as electronic boards to be integrated within a final product. This product must have an enclosure to protect it from dust, humidity and other environmental interactions. If the product is to be used outside, the enclosure must have an IP-65 rating, at the minimum.
- Do not place Waspmote in contact with metallic surfaces; they could cause short-circuits which will permanently damage it.
Specific:

- Reset and ON/OFF button: Handle with care, do not force activation or use tools (pliers, screwdrivers, etc) to handle it.
- Battery: Only use the original lithium battery provided with Waspmote.
- Solar panel connection: Only use the connector specified in the Power supplies section and always respect polarity.
- Lithium battery connection: Only use the connector specified in the Battery section and always respect polarity.
- GPS board connection: Only use the original Waspmote GPS board.
- XBee module connection: Waspmote allows the connection of any module from the XBee family, respect polarity when connecting (see print).
- Antenna connections: Each of the antennas that can be connected to Waspmote (or to its GPS board) must be connected using the correct type of antenna and connector in each case, or using the correct adaptors.
- USB voltage adaptors: To power and charge the Waspmote battery, use only the original accessories: 220V AC – USB adaptor and 12V DC (car cigarette lighter) – USB adaptor

Usage and storage recommendations for the batteries:

The rechargeable, ion-lithium batteries, like the ones provided by Libelium (capacities of 6600 mAh), have certain characteristics which must be taken into account:

- Charge the batteries for 24 hours before a deployment. The aim is to have the charge of the batteries at 100% of their capacity before a long period in which they must supply current, but it is not necessary to improve the performance.
- It is not advised to let the charge of the batteries go below 20% of capacity, since they suffer stress. Thus, it is not advised to wait for the battery to be at 0% to charge it.
- Any battery self-discharges: connected to Waspmote or not, the battery loses charges by itself.
- Maximum capacity loss: as the charge and discharge cycles happen, the maximum charge capacity is reduced.
- Batteries work better in cool environments: their performance is better at 10 ºC than at 30 ºC.
- At temperatures below 0 ºC, batteries can supply current (discharge), but the charge process cannot be done. In particular:
  - discharge range = [-10, 60] ºC
  - charge range = [0, 45] ºC

It is not recommended to have the non-rechargeable batteries (13000, 26000, 52000 mA·h) connected to Waspmote when the USB cable is connected too. The reason is, Waspmote will try to inject current in them if the USB is connected. This is dangerous for the good working of a non-rechargeable battery. It could be damaged or even damage Waspmote. That is to say, when you need to upload code to Waspmote via USB, disconnect the battery if it is non-rechargeable. That applies to Waspmote OEM, but not to the Plug & Sense! line, since its hardware is modified to avoid this.
2.3. Assembly

- Connect the antenna to the 6LoWPAN module

- Place 6LoWPAN module in Waspmote

- Place the Ethernet Module in the Expansion board
- Place the Expansion Board + Ethernet Module in Waspmote

- Connect the antenna in the GPS module

- Place the GPS module in Waspmote
- Connect the battery in Waspmote

- Connect the sensor board

- Switch it on
- Waspmote battery disconnection

Use the pick supplied by Libelium in order to disconnect Waspmote battery.

1. Insert the pick on the slot of the battery connector and pull straight out. Do not pull the battery cables.
- Battery handling instructions
  In order to prevent from cable breaking, avoid leaving battery freely suspended.

Use a nylon clamp in order to attach battery to Waspmote.
2.4. Powering the Nodes

2.4.1. Battery

The battery included with the Waspmote Mote Runner Kits is a Lithium-ion battery (Li-Ion) with 3.7V nominal voltage and 6600mAh of capacity.

Waspmote has a control and safety circuit which makes sure the battery charge current is always adequate.

![Battery connection](image)

Battery connection

The figure below shows the connector in which the battery is to be connected. The position of the battery connector is unique, therefore it will always be connected correctly (unless the connector is forced).
2.4.2. USB

Figure: Mini USB connector

Waspmote’s USB power sources are:

- USB to PC connection
- 110/220V to 5V USB Connection

**Note:** The nodes can not work with the radio and the USB plugged to a PC. The right way of powering them up once the firmware has been uploaded is by using the 110/220V to 5V USB Power Adaptor supplied in the kit.

Obviously they can work too by using just the batteries provided without the need of connecting any cable (always they are with load).

The models supplied by Libelium are shown below:

Figure: 110/220V to 5V USB Power Adaptor
2.4.3. Solar Panel

The solar panel must be connected using the cable supplied.

Both the mini USB connector and the solar panel connector allow only one connection position which must be respected without being forced into the incorrect position. In this way connection polarity is respected.

Solar panels up to 12V are allowed. The maximum charging current through the solar panel is 280mA.
The models supplied by Libelium are shown below:

- **Rigid Solar Panel**
  
  7V - 500mA

- **Flexible Solar Panel**
  
  7.2V - 100mA
3. Waspmote’s Hardware

3.1. Modular Architecture

Waspmote is based on a modular architecture. The idea is to integrate only the modules needed in each device. These modules can be changed and expanded according to needs.

The modules available for integration in Waspmote are categorized in:

- ZigBee/802.15.4 modules (2.4GHz, 868MHz, 900MHz). Low and high power.
- GSM/GPRS Module (Quadband: 850MHz/900MHz/1800MHz/1900MHz)
- 3G/GPRS Module (Tri-Band UMTS 2100/1900/900MHz and Quad-Band GSM/EDGE, 850/900/1800/1900 MHz )
- GPS Module
- Sensor Modules (Sensor boards)
- Storage Module: SD Memory Card

3.2. Specifications

- **Microcontroller:** ATmega1281
- **Frequency:** 14.7456 MHz
- **SRAM:** 8KB
- **EEPROM:** 4KB
- **FLASH:** 128KB
- **SD Card:** 2GB
- **Weight:** 20gr
- **Dimensions:** 73.5 x 51 x 13 mm
- **Temperature Range:** [-10ºC, +65ºC]
3.3. Block Diagram

Data signals:
Power signals:

Figure: Waspmote block diagrams – Power signals

3.4. Electrical Data

Operational values:
- Minimum operational battery voltage 3.3 V
- Maximum operational battery voltage 4.2 V
- USB charging voltage 5 V
- Solar panel charging voltage 6 - 12 V
- Battery charging current from USB 100 mA (max)
- Battery charging current from solar panel 280 mA (max)

Absolute maximum values:
- Voltage in any pin [-0.5 V, +3.8 V]
- Maximum current from any digital I/O pin 40 mA
- USB power voltage 7 V
- Solar panel power voltage 18 V
- Charged battery voltage 4.2 V
3.5. I/O

Waspmote can communicate with other external devices through the using different input/output ports.

**Sensor connector:**

- ANALOG
- DIGITAL 8
- DIGITAL 6
- DIGITAL 4
- DIGITAL 2
- RESERVED
- ANALOG 6
- ANALOG 4
- ANALOG 2
- 3V3 SENSOR POWER
- GPS POWER
- SDA
- 3V3 SENSOR POWER
- GND
- DIGITAL 7
- DIGITAL 5
- DIGITAL 3
- DIGITAL 1
- ANALOG 7
- ANALOG 5
- ANALOG 3
- ANALOG 1
- 5V SENSOR POWER
- SCL
- GND
- ANALOG 6
- ANALOG 7
- SCK
- RXD1
- TXD1
- MOSI
- MISO
- 3V3 SENSOR
- 3V3 SENSOR

**Auxiliary SPI-UART connector:**

- AUX SERIAL 1TX
- AUX SERIAL 1RX
- AUX SERIAL 2TX
- AUX SERIAL 2TX
- BATTERY
- GND
- SCK
- RXD1
- TXD1
- 3V3 SENSOR POWER
- MOSI
- MISO
- 3V3 SENSOR
3.5.1. Analog

Waspmote has 7 accessible analog inputs in the sensor connector. Each input is directly connected to the microcontroller. The microcontroller uses a 10 bit successive approximation analog to digital converter (ADC). The reference voltage value for the inputs is 0V (GND). The maximum value of input voltage is 3.3V which corresponds with the microcontroller’s general power voltage.

There are two ways to obtain the input values, asynchronously and synchronously.

In both ways, the channels to be read must be opened with the method open() of the ADC class, indicating them inside the channel map used as parameter.

```java
static byte ADC_1 = 1;
static ADC adc = new ADC();
chmap = chmap | (int) (1 << ADC_1);
adc.open(chmap, GPIO.NO_PIN, 0, 0);
```

To obtain input values asynchronously, the method read() of the ADC class and the callback system might be used:

```java
adc.setReadHandler(new DevCallback(null) {
    public int invoke(int flags, byte[] data, int len, int info, long time) {
        return <Class>.callbackMethod(flags, data, len, info, time);
    }
});
adc.read(Device.ASAP, 1, 0);
```

To obtain input values synchronously, the method readChannel(byte channel) of the ADC class might be used.

```java
int value = adc.readChannel(ADC_4);
```

*Note that the code in this section is Java code.*

3.5.2. Digital

Waspmote has digital pins which can be configured as input or output depending on the needs of the application. The voltage values corresponding to the different digital values would be:

- 0V for logic 0
- 3.3V for logic 1

There are two ways to control digital pins, asynchronously and synchronously.

In both ways, a GPIO instance must be created and opened as follows:

```java
GPIO gpio = Gpio.getInstance(); //Unique instance of GPIO object
gpio.open();
```

For writing in the digital pins the first time, the method configureOutput(byte pin, byte mode) of the class GPIO must be used, and the following times, the method doPin(byte ctrl, byte pin) is the correct:

```java
gpio.configureOutput(WASPMOTE.PIN_DIGITAL1, GPIO.OUT_SET);
gpio.doPin(GPIO.CTRL_CLR, WASPMOTE.PIN_DIGITAL1);
```

For reading from the digital pins asynchronously, the method read() from the GPIO class and the callback system must be used:

```java
gpio.configureInput(WASPMOTE.PIN_DIGITAL1, GPIO.IRQ_DISABLED, (byte) 0);
gpio.setEventHandler(new DevCallback(null) {
    public int invoke(int flags, byte[] data, int len, int info, long time) {
        return <Class>.callbackMethod(flags, data, len, info, time);
    }
});
```
For reading from the digital pins synchronously, the method `doPin()` with the `CTRL_READ` control constant of the class `GPIO` must be used:

```java
gpio.configureInput(WASPMOTE.PIN_DIGITAL1, GPIO.IRQ_DISABLED, (byte) 0);
int result = gpio.doPin(GPIO.CTRL_READ, WASPMOTE.PIN_DIGITAL1);
```

Note that the code on this section is Java code and assumes that `com.libelium.common` is imported.

### 3.5.3. UART

Transmission to the UART has to be done using Cdev class as follows:

```java
static{
    CDev xbee = new CDev();
    byte[] settings = csr.allocaByteArray((byte)4);
    settings[0] = WASPMOTE.UART0; // UART0, plain (no CRC)
    settings[1] = WASPMOTE.UART_FRAME_DATA_BIT_8 |
                  WASPMOTE.UART_FRAME_STOP_BIT_1 |
                  WASPMOTE.UART_FRAME_PARITY_NONE; // frame (8-bits), no parity, 1-bit stop
    settings[2] = WASPMOTE.UART_BAUD_4800; // baud high (4800)
    xbee.open(WASPMOTE.DID_UART, settings, 0, 4);
    xbee.setTxHandler(new DevCallback(null) {
        public int invoke(int flags, byte[] data, int len, int info, long time) {
            return <Class>.callbackMethod(flags, data, len, info, time);
        }
    });
    xbee.transmit(Device.ASAP, csr.s2b("+++"), (byte)0, 1, 0);
}
```

### 3.5.4. I2C

The I2C communication bus is also used in Waspmote where two devices are connected in parallel: the accelerometer and the RTC. In all cases, the microcontroller acts as master while the other devices connected to the bus are slaves.

See complete I2C documentation at:

```markdown
```

Note that mrsh should be running in order to access the mentioned link.

### 3.5.5. SPI

The SPI port on the microcontroller is used for communication with the micro SD card. All operations using the bus are performed clearly by the specific library. The SPI port is also available in the SPI/UART connector.

See complete SPI documentation at:

```markdown
```

Note that mrsh should be running in order to access the mentioned link.
3.5.6. USB

USB is used in Waspmote for communication with a computer or compatible USB devices. This communication allows the microcontroller’s program to be loaded. For USB communication, microcontroller’s UART0 is used. The FT232RL chip carries out the conversion to USB standard.

3.6. Real Time Clock-RTC

Waspmote has a built in Real Time Clock – RTC, which keeps it informed of the time. This allows Waspmote to be programmed to perform time-related actions such as:

“Sleep for 1h 20 min and 15sec, then wake up and perform the following action”

Or even programs to perform actions at absolute intervals, e.g.:

“Wake on the 5th of each month at 00:20 and perform the following action”

All RTC programming and control is done through the I2C bus.

Alarms:

Alarms can be programmed in the RTC specifying day/hour/minute/second. That allows total control about when the mote wakes up to capture sensor values and perform actions programmed on it. This allows Waspmote to be in the saving energy modes (Deep Sleep and Hibernate) and makes it wake up just at the required moment.

As well as relative alarms, periodic alarms can be programmed by giving a time measurement, so that Waspmote reprograms its alarm automatically each time one is triggered.

The RTC chosen is the Maxim DS3231SN, which operates at a frequency of **32.768Hz** (a second divisor value which allows it to quantify and calculate time variations with high precision).

The DS3231SN is one of the most accurate clocks on the market because of its internal compensation mechanism for the oscillation variations produced in the quartz crystal by changes in temperature (Temperature Compensated Crystal Oscillator – TCXO).

Most RTCs on the market have a variation of ± 20ppm which is equivalent to a 1.7s loss of accuracy per day (10.34min/year), however, the model chosen for Waspmote has a loss of just ± 2ppm, which equates to variation of 0.16s per day (1min/year).

![Uncompensated variation curve](image1.png) ![Compensated variation curve](image2.png)

Figure on the left shows the temperature variation curve in a typical commercial clock, and in figure on the right that for the DS3231SN model built into Waspmote. As can be seen, variations in accuracy are practically zero at room temperature and minimal when moved to the ends of the temperature scale.
The recalibration process of the oscillation crystal is carried out thanks to the data received by the RTC’s *internal temperature sensor*. The value of this digital sensor can be accessed by Waspmote through the I2C bus, which lets it know the **temperature of the board** at anytime in the range of **-40ºC to +85ºC** with an accuracy of 0.25°C. For more information about the acquisition of this value by the microprocessor, see the section “Sensors in Waspmote → Temperature”.

**Note:** the RTC’s *internal temperature sensor is only meant for the time derive compensation, but not for common air temperature sensing (we advise our Sensor Boards for that).*

The RTC is powered by the battery. When the mote is connected, the RTC is powered through the battery, but take into account that if the battery is removed or out of load, then time data will be not maintained. That is why we suggest to use RTC time like ‘relative’ and not ‘absolute’ (see Programming Guide for more info).

A coin or button battery is no longer needed. They had a limited life and therefore Waspmote can now have a much longer power life expectancy. This is so because the RTC is powered from the “main” battery which has a much bigger charge.

See complete documentation at section “RTC”.

### 3.7. LEDs

![Visual indicator LEDs](image)

#### Charging battery LED indicator
A red LED indicating that there is a battery connected in Waspmote which is being charged, the charging can be done through a mini USB cable or through a solar panel connected to Waspmote. Once the battery is completely charged, the LED switches off automatically.

#### LED 0 – programmable LED
A green indicator LED is connected to the microcontroller. It is totally programmable by the user from the program code. In addition, the LED 0 indicates when Waspmote resets, blinking each time a reset on the board is carried out.

#### LED 1 – programmable LED
A red indicator LED is connected to the microcontroller. It is totally programmable by the user from the program code.

#### USB Power LED indicator
A green LED which indicates when Waspmote is connected to a compatible USB port either for battery charging or programming. When the LED is on it indicates that the USB cable is connected correctly, when the USB cable is removed the LED will switch off automatically.
3.8. Communication Modules

3.8.1. 6LoWPAN Radios

6LoWPAN Radio (2.4GHz)
- Chipset: AT86RF231
- Frequency: 2.4GHz
- Link Protocol: IEEE 802.15.4
- Usage: Worldwide
- Sensitivity: -101dBm
- Output Power: 3dBm
- Encryption: AES 128b

6LoWPAN Radio (868MHz)
- Chipset: AT86RF212
- Frequency: 868MHz
- Link Protocol: IEEE 802.15.4
- Usage: Europe
- Sensitivity: -110dBm
  Output Power: 10dBm
- Encryption: AES 128b

3.8.2. Ethernet Module
- Chipset: W5100
- Protocol: Ethernet IPv4
- Physical: 100BASE-TX
- Services: TCP/IP, UDP/IP / ICMP
- Internal Buffer: 16KB
4. Architecture and System

IBM Mote Runner is a run-time platform and development environment for wireless sensor networks (WSN) currently under development at the IBM Zurich Research Laboratory, Switzerland. It consists of an on-mote environment, the so-called “firmware,” with some standard libraries, an off-mote environment (including an edge server facilitating communication between the WSN and the outside world, a command shell permitting high-level control of the WSN, and source-level debugger for refinement of the application code running inside the WSN), and a set of development tools such as a command-line compiler and graphical integrated development environment (IDE).

Figure: Overview of Mote Runner Ecosystem

4.1. Concepts

On the mote platform the Mote Runner firmware provides a run-time environment, which incorporates a virtual machine (VM) for executing byte codes and operating system (OS) to organize access to different devices and to schedule the various activities of applications.

Those byte codes are generated compiling an application with the Mote Runner Compiler. Mote Runner applications can be written in C# or Java.

In general applications are shielded from the underlying hardware and management functionality is provided by the VM/OS. The Mote Runner VM provides only a single thread of execution. However, multiple applications can be running on top of the VM/OS at the same time. As a general rule, synchronous calls or busy looping will block the possible execution of other VM applications, and may not be energy efficient.
Managing assemblies on the motes is done through the sonoran gateway, a process running on a computer. This process can establish communication also with some other processes as for example scripts, browser or Eclipse IDE acting as a bridge between motes and the outside world.

Sonoran is a Javascript based programming and management environment for Mote Runner. It uses the V8 Javascript engine which is extended by a number of native functions (e.g. file and network IO) to service the Sonoran Javascript framework. On top of the core and Sonoran javascript framework, a number of interfaces exist allowing the interaction with Sonoran and motes. Sonoran features a command-line-shell (either on stdin/stdout or in a telnet session) and a HTTP interface (used by the Comote API) to setup and communicate with a network of motes. Additionally, Sonoran can execute user-defined scripts and applications building upon the base Sonoran functionality.

See complete Sonoran documentation at:
http://localhost:5000/doc/system/index.html#system_scriptenv_0

Note that mrsh should be running in order to access the mentioned link

4.2. Programming modes

4.2.1. Reactive Programming (Asynchronous)

Applications implemented on top of the VM follow a reactive programming model. An application registers callbacks with operating services which will be invoked on certain events. An event starts with calling a specific method in the VM which probably will in turn call other methods. The application code may call the operating system to request certain actions but none of these calls will block the application. Long lasting operations create completion events leading again to callbacks. All VM callbacks execute to completion and will not interrupt other callbacks.

Such a reactive programming model is very resource conservative. The image below shows the basic interactions between the VM and the operating system. The central abstraction of the OS are devices. Devices represent sensors, actuators, communication devices like radio or serial and USB connections.

![Reactive programming flow](image-url)

*Figure: Reactive programming flow*
4.2.2. Imperative programming (Synchronous)

Despite all applications can be programmed following a reactive programming model, some of the Mote Runner APIs offer synchronous functions for accessing some devices like:

- ADC
- GPIO
- I2C
- SPI

It is important to notice that synchronous calls will block the possible execution of other VM applications, and may not be energy efficient.

Figure: Imperative programming flow

4.3. Boot time connections

In order to connect with a Sonoran process running on a computer different kind of connections can be established.

As a general rule, at boot time, the Mote Runner OS scans for available communication connections. In case of the Waspmote the Mote Runner firmware will try the following connections in the given order.

1. **Ethernet**: found when the Ethernet extension board is attached to the Waspmote (Note the EEPROM of the Waspmote requires an IP configuration).
2. **Serial (LIP) via the USB port**: found when the Waspmote is attached to the PC via a USB cable, the mrsh started and, a connection to the mote exists in mrsh, e.g. `mote-create -p /dev/ttyUSB0`.
3. **WLIP via the Radio module (both 2.4 GHz and 868 MHz)**: found when the RF2xx radio extension board is attached.

If one connection is found it will be selected as the default for communication via the LIP API and further possible connections are not scanned:

*Note: It is not possible to operate both a Mote Runner Radio device (with the RF2xx radio boards) and a LIP connection via USB.*
**Note:** If an application opens the Radio device in the static initializer, then WLIP (which itself needs to open the Radio) will not be started up by the OS.

At boot time the VM/OS will execute the static class initializers for all installed applications, one by one. If the class initializer for one application fails, e.g., because of an uncaught exception, the system will restart. Thus, it is always good practice to catch all possible exceptions in the class initializers.

An application can query the state of the LIP connection by using `LIP.isAvailable` call provide by the LIP API. Querying the state of LIP can be safely performed on the `onSystemInfo` callback event, which informs applications that the OS finished the boot sequence.

### 4.3.1. USB connection and Radio Modules

The USB connection for data transmission uses the same UART as the radio modules socket. For this reason data transmission through USB can't be done at the same time as data transmission through radio modules.

There is no problem with using the radio modules and the USB at the same time if the USB is only used to power the mote or charge the battery.

### 4.4. Networking

#### 4.4.1. IPv4-UDP

#### 4.4.1.1. IP configuration through MOMA

Using the moma-ipv4 command the IPv4 configuration can be queried and set. Without any parameters the command would print the current settings. All moma commands should be executed on Mote Runner shell (mrsh).

```bash
> moma-ipv4
IPv4 address: 192.168.1.100
IPv4 gateway: 192.168.1.1
UDP port: 4369 (0x1111)
```

#### 4.4.1.2. IP configuration through source code

The IPv4 configuration, similar to the EUI configuration, can also be changed and queried programatically using the `getParam/setParam` calls in the Mote class provided by the saguaro-system API.

```java
// set up my own IP address
byte[] myIPaddr = {192,168,1,100};
Mote.setParam(Mote.IPv4_ADDRESS, myIPaddr, 4);
// set up my gateway address (used for routing)
byte[] myIPgw = {192,168,1,1};
Mote.setParam(Mote.IPv4_GATEWAY, myIPgw, 4);
// set the UDP server port, which would be used by the udp-create command
byte[] myUDPport = {0x11,0x11}; // little endian = corresponds to 4369
Mote.setParam(Mote.IPv4_UDPPORT, myUDPport, 2);
```

See complete Mote class documentation at:


Note that mrsh should be running in order to access the mentioned link.
4.4.2. IPv6-UDP

See complete IPv6 documentation at:
“6LoWPAN Network” section

4.4.3. Networking between sonoran and motes

4.4.3.1. Mote Runner Shell (mrsh) Sockets

For simulated mote as well as a hardware motes, which are connected via LIP to mrsh, LIP/UDP messages are routed through the Javascript environment and forwarded accordingly. The socket commands can be used on the ‘mrsh’ command line to send UDP messages to motes and to print received UDP messages. The following command creates a new socket in the shell named “SOCK”.

```bash
> sock-bind SOCK
```

The following creates a new simulation process and new mote.

```bash
> sag-start; mote-create
```

The following sends a message to mote ‘a0’ and ‘mrsh’ prints the immediately following message on the console.

```bash
> a0 sock-send SOCK 0 03
16:27:629: DSTPORT 206 SRCPORT 0 SRC 02-00-00-00-00-DF-60-00 00: 03 00 00 03 49 0E 73 61 67 75 61 72 6F 2D ....I,.saguaro-s ...
```

Destination port 0 of the command is the MOMA port and the bytes sent form the `moma-list` message as described in the MOMA section.

4.4.3.2. Mote Runner Shell (mrsh) as a UDP/LIP bridge

The `udp-bridge` command in mrsh can be used to exchange LIP messages between an external host and a connected mote both simulated and hardware.

The following command opens a UDP port which is used to forward UDP messages to a mote, in this case the l0 mote.

```bash
> l0 udp-bridge 61234
```

LIP messages received from mrsh on port 61234 are forwarded to the mote, host and source port in the LIP message modified. Messages from the mote to the UDP bridge are forwarded to the external host (i.e. the host the UDP bridge received the last message from).

4.4.4. Networking on mote applications

For efficiency reasons, as a general rule, Mote Runner uses UDP-based communication to and from motes. Moreover, applications have a uniform interface for communication using LIP which abstracts from the actual communication channel (be it serial, Ethernet, or WLIP).

The format of LIP messages is as follows

- [4] IPv4 address: source, when a message is received, destination when a message is sent
- [2] UDP port: source, when a message is received, destination when a message is sent
- [1] Application port (used when receiving a message)

--- payload

To receive LIP messages, an application needs to either open a port using the `LIP.open` call, or simply register a callback using the `Assembly.setOnData` call.
A mote application can send a LIP message using `LIP.send`. In the header, the IPv4 address and UDP port can be specified:

```java
// set the IP address
byte [] udpDestination = {192, 168, 1, 2, 0, 0};
// set the UDP port
Util.set16le(udpDestination,4,61234);
// send the message
LIP.send(udpDestination,6,message,0,(uint)message.length);
```

Note that the 7-byte LIP header can be prefixed by an additional header depending on the communication channel. The offset of the LIP application port can be queried using the `LIP.getPortOff` call.

See complete LIP class documentation at:


Note that mrsh should be running in order to access the mentioned link

### 4.5. Timers

The hardware timers available on the underlying hardware platform are abstracted using the Mote Runner.

The granularity of the timers are the underlying MCU ticks. In case of the Waspmote, a tick is roughly equivalent with 1 us. However, as underlying hardware may change, applications should use the corresponding conversion functions from SI units to hardware ticks when dealing with timers or scheduling of tasks.

```java
Timer timer = new Timer();

//Set the callback for this timer
timer.setCallback(new TimerEvent(null){
    public void invoke(byte param, long time){
        <name_of_the_class>.name_of_callback_function(param,time);
    }
});

// convert 2 seconds to the platform ticks
INTERVAL = Time.toTickSpan(Time.SECONDS, 2);

// set a new alarm in 2 seconds from now
timer.setAlarmBySpan(INTERVAL);
```

See complete Timer API at:


Note that mrsh should be running in order to access the mentioned link
4.6. Watchdog

In general hardware platforms provide a watchdog mechanism. In Mote Runner applications can enable this mechanism using the `Mote.watchdog` API call.

By default, the watchdog is not started. Once, started the watchdog must be restarted periodically.

See complete Watchdog documentation at:

Note that mrsh should be running in order to access the mentioned link

4.7. RTC

The `com.libelium rtc` is the responsible of managing the use of the real time clock attached in the motes.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```
> cd /API path/com/libelium/rtc
> mrc --assembly=rtc-1.0 --system=waspmote --ref=../common/common-1.0 RTC.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```
> cd/API path/com/libelium/rtc
> moma-load rtc
```

These are the methods implemented in the library:

```java
public void setDate(byte year, byte month, byte date, byte day, byte hours, byte minutes, byte seconds);
public byte[] getDate();
public void setAlarm1(Callback cb, byte mode, boolean offset, byte date, byte day, byte hours, byte minutes, byte seconds);
public byte[] getAlarm1();
public void setAlarm2(Callback cb, byte mode, boolean offset, byte date, byte day, byte hours, byte minutes);
public byte[] getAlarm2();
public static void disableAlarm(byte alarm);
public long getTemperature();
```
And a simple example of the use of them:

```java
static {

    static RTC rtc = RTC.getInstance(); //Unique instance of the RTC Object

    //Set the actual date of the RTC to Saturday, 2010-06-04 at 02:03:50
    rtc.setDate((byte)10, (byte)6, (byte)4, (byte)7, (byte)2, (byte)3, (byte)50);

    //Set the alarm 1 to be fired every second (see the alarm modes on
    //the documentation of the methods)
    rtc.setAlarm1(new Callback(null){
        public int invoke(byte[] result, int len) {
            return PruebasRTC.alarmCallback(result, len);
        }
    }, (byte)Constants.RTC_ALM1_MODE1, false, (byte)0, (byte)0, (byte)0, (byte)0);

    //Set the alarm 2 to be fired on the minute 1 of every hour
    rtc.setAlarm2(new Callback(null){
        public int invoke(byte[] result, int len) {
            return PruebasRTC.alarmCallback2(result, len);
        }
    }, (byte)Constants.RTC_ALM2_MODE2, false, (byte)0, (byte)0, (byte)0, (byte)1);

    //Get the actual info of the alarm 2 and print it on the console
    byte[] alarma2 = rtc.getAlarm2();
    Logger.appendString(csr.s2b("minute: ");
    Logger.appendInt(alarma2[0]);
    Logger.appendString(csr.s2b("hour: ");
    Logger.appendInt(alarma2[1]);
    Logger.appendString(csr.s2b("date: ");
    Logger.appendInt(alarma2[2]);
    Logger.flush(Mote.INFO);
}

public static int alarmCallback(byte[] result, int len){
    //Get the actual Date stored on the RTC and print it on the console
    byte[] date = rtc.getDate();
    Logger.appendString(csr.s2b("second: ");
    Logger.appendInt(date[0]);
    Logger.appendString(csr.s2b("minute: ");
    Logger.appendInt(date[1]);
    Logger.appendString(csr.s2b("hour: ");
    Logger.appendInt(date[2]);
    Logger.appendString(csr.s2b("date: ");
    Logger.appendInt(date[3]);
    Logger.appendString(csr.s2b("month: ");
    Logger.appendInt(date[4]);
    Logger.appendString(csr.s2b("year: ");
    Logger.appendInt(date[5]);
    Logger.flush(Mote.INFO);
    LED.setState((byte)0, (byte)(LED.getState((byte)0)^1));
    return 0;
}

public static int alarmCallback2(byte[] result, int len){
    LED.setState((byte)1, (byte)(LED.getState((byte)1)^1));
    return 0;
}
```

Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:

4.8. Interruptions

The low-level hardware interruptions are entirely handled by the VM/OS. Applications will only receive corresponding callback events for which they have registered. In the asynchronous callback, the applications deal with the processing of the corresponding data, generated by the interrupts. As a general rule, synchronous calls or busy looping will block the possible execution of other VM applications, and may not be energy efficient.

4.9. Sleep mode

To save energy, based on the tasks ahead, the Mote Runner VM/OS decides on the best sleep mode for the underlying hardware system, including sensor and communication devices. While, applications do not have a direct API access to the hardware sleep modes of the MCU or the Radio device, the applications do influence the sleep mode by their queued tasks. The following rules are used by the VM/OS to decide on the sleep mode.

- **SLEEP**: the next task is scheduled far enough ahead (no other task which can generate a callback); all hardware modules (MCU, sensors, radios, etc.) and will be powered off.
- **IDLE**: no task which can potentially generate a callback, e.g., a receive task from a communication device such as the radio, Ethernet, or USB.
- **ACTIVE**: whenever VM callbacks and OS code execute.

For example, if there is only one task to be performed a few seconds ahead and no other task is scheduled in the meantime, the OS will try to put the system into the most efficient sleep mode. As another example, if a communication device such as the radio has queued a reception task.
5. Sensors

5.1. Internal Sensors

5.1.1. Temperature

The Waspmote RTC (DS3231SN from Maxim) has a built in internal temperature sensor which it uses to recalibrate itself. Waspmote can access the value of this sensor through the I2C bus.

The sensor is shown in a 10-bit two's complement format. It has a resolution of 0.25º C. The measurable temperature range is between -40ºC and +85ºC.

As previously specified, the sensor is prepared to measure the temperature of the board itself and can thereby compensate for oscillations in the quartz crystal it uses as a clock. As it is a sensor built in to the RTC, for any application that requires a probe temperature sensor, this must be integrated from the micro's analog and digital inputs, as has been done in the case of the sensor boards designed by Libelium.

See complete temperature documentation at section "RTC"
5.1.2. Accelerometer

Waspmote has a built-in acceleration sensor LIS3331LDH STMicroelectronics which informs the mote of acceleration variations experienced on each one of the 3 axes (X, Y, Z).

The integration of this sensor allows the measurement of acceleration on the 3 axes (X, Y, Z), establishing 4 kinds of events: Free Fall, inertial wake up, 6D movement, and 6D position, which are explained in the Interruptions Programming Guide.

The LIS331DLH has dynamically user selectable full scales of $\pm 2g/\pm 4g/\pm 8g$ and it is capable of measuring accelerations with output data rates from 0.5 Hz to 1 kHz.

The device features ultra-low-power operational modes that allow advanced power saving and smart sleep to wake-up functions.

The accelerometer has 7 power modes, the output data rate (ODR) will depend on the power mode selected. The power modes and output data rates are shown in this table:

<table>
<thead>
<tr>
<th>Power mode</th>
<th>Output data rate (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power down</td>
<td>--</td>
</tr>
<tr>
<td>Normal mode</td>
<td>1000</td>
</tr>
<tr>
<td>Low-power 1</td>
<td>0.5</td>
</tr>
<tr>
<td>Low-power 2</td>
<td>1</td>
</tr>
<tr>
<td>Low-power 3</td>
<td>2</td>
</tr>
<tr>
<td>Low-power 4</td>
<td>5</td>
</tr>
<tr>
<td>Low-power 5</td>
<td>10</td>
</tr>
</tbody>
</table>

The LIS331DLH has dynamically user selectable full scales of $\pm 2g/\pm 4g/\pm 8g$ and it is capable of measuring accelerations with output data rates from 0.5 Hz to 1 kHz.
This accelerometer has an auto-test capability that allows the user to check the functioning of the sensor in the final application. Its operational temperature range is between -40ºC and +85ºC.

The accelerometer communicates with the microcontroller through the I2C interface. The pins that are used for this task are the SCL pin and the SDA pin, as well as another INT pin to generate the interruptions.

The accelerometer has 4 types of event which can generate an interrupt: free fall, inertial wake up, 6D movement and 6D position.

Some figures with possible uses of the accelerometer are shown below:

**Rotation and Twist:**

![Rotation and Twist Diagram](image)

**Vibration:**

![Vibration Diagram](image)
Acceleration:

Free fall:

Free fall of objects in which it is installed:
5. Sensor boards

5.2. Common library

The com.libelium.common library, which includes Callback.java, Common.java, Constants.java, Gpio.java and Interrupts.java, is a library of methods and utilities used by the rest of the board libraries.

The Constants.java class includes the global constants of the API.

The Gpio.java class manages general purpose inputs and outputs in the API.

The Interrupts.java class handles interrupts, enabling, capturing and treating them.

The Common.java class contains methods that all the libraries of the API use.

The Callback.java is an instantiable object that will be responsible for containing the callback method that will be called when an asynchronous action has been finished.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```bash
> cd /API path/com/libelium/common
> mrc --assembly=common-1.0 --system=waspmote Constants.java Common.java Gpio.java Interrupts.java Callback.java
```

To upload the compiled assembly to the Waspmote, runs as follows:

```bash
> cd/API path/com/libelium/common
> moma-load common
```

For further information about building and loading programs in mote runner, see the section "Application development".

This library must be referenced when building any of the other libraries and must be loaded in the waspmote for a proper working of the system.
5.2.2. Smart Cities Library

The `com.libelium.smartcities` is the responsible of managing the use of the sensor board called Smart Cities.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```bash
> cd /API path/com/libelium/smartcities
> mrc --assembly=smartcities-1.0 --system=waspmote --ref=../common/common-1.0 SmartCities.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```bash
> cd/API path/com/libelium/smartcities
> moma-load smartcities
```

Here are the necessary steps to read a value from a sensor:

1. **Create the object of the SmartCities library class and switch on the same:**

   ```java
   static smartCities sc = new SmartCities();
   sc.ON();
   ```

2. **Set the state of the sensor that is gonna be read:**

   ```java
   sc.setSensorMode(byte mode, int sensor);
   ```

   Where mode constants are:

   - `Constants.SENS_ON` : turn on the sensor
   - `Constants.SENS_OFF` : turn off the sensor

   And sensor unique id constants are:

   - `Constants.SENS_CITIES_DUST` : Dust Sensor
   - `Constants.SENS_CITIES_LD` : Linear Displacement Sensor
   - `Constants.SENS_CITIES_AUDIO` : Noise Sensor
   - `Constants.SENS_CITIES_HUMIDITY` : Humidity Sensor
   - `Constants.SENS_CITIES_TEMPERATURE` : Temperature Sensor
   - `Constants.SENS_CITIES_ULTRASOUND` : Ultrasound Sensor
   - `Constants.SENS_CITIES_LDR` : Luminosity Sensor
   - `Constants.SENS_CITIES_CD` : Crack detection Sensor
   - `Constants.SENS_CITIES_CP` : Crack propagation Sensor

3. **Call the read method and store the value returned:**

   ```java
   long result = sc.readValue(int sensor);
   ```

A simple complete example of the use of this library for reading the current sensor:

```java
static SmartMetering sm = new SmartMetering();
sc.ON();
sc.setSensorMode(Constants.SENS_ON, Constants.SENS_CITIES_AUDIO);
long noise = sc.readValue(0);
Logger.appendString(csr.s2b("Noise measurement: "));
Logger.appendHexLong(csr.toLong(noise,()
Logger.flush(Mote.INFO);
```
This library also provides a system of interrupts that may be caused when a sensor detects that the value read is higher than an indicated threshold.

The steps to configure that threshold are the following:

1. Set the threshold value of the sensor:
   
   ```
   sc.setThreshold(int sensor, long threshold);
   ```

2. Attach the interrupts and set the callback function that will be called when the interrupt is fired:
   
   ```
   sc.attachInt(new Callback(null) {
       public int invoke(byte[] buf, int len) {
           return callbackInterrupt(buf, len);
       }
   });
   ```

3. Then, in the `callbackInterrupt` method, discriminate the sensor that caused the interruption:
   
   ```
   byte flags = sc.loadInt();
   if((flags & <sensor constant>) > 0))
   ```

A simple complete example of the use of this library for setting a threshold in the noise sensor and capture the interruption caused:

```java
static {
    sc.ON();
    sc.setSensorMode(Constants.SENS_ON, Constants.SENS_CITIES_AUDIO);
    sc.setThreshold(Constants.SENS_CITIES_AUDIO, Float.make(10, (byte)-1));
    sc.attachInt(new Callback(null) {
        public int invoke(byte[] buf, int len) {
            return callbackInterrupt(buf, len);
        }
    });
}

private static int callbackInterrupts(byte[] buf, int len){
    byte flags = sc.loadInt();
    if((flags & Constants.SENS_CITIES_AUDIO)>0){
        Logger.appendString(csr.s2b("Noise interruption"));
        Logger.flush(Mote.INFO);
    }
}
```

Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:

5.2.3. Smart Metering Library

The `com.libelium.smartmetering` is the responsible of managing the use of the sensor board called Smart Metering.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```
> cd /API path/com/libelium/smartmetering
> mrc --assembly=smartmetering-1.0 --system=waspmote --ref=../common/common-1.0 SmartMetering.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```
> cd/API path/com/libelium/smartmetering
> moma-load smartmetering
```

Here are the necessary steps to read a value from a sensor:

1. **Create the object of the SmartMetering library class and switch on the same:**
   ```
   static SmartMetering sm = new SmartMetering();
   sm.ON();
   ```

2. **Set the state of the sensor that is gonna be read:**
   ```
   sm.setSensorMode(byte mode, int sensor);
   ```

Where mode constants are:

- Constants.SENS_ON : turn on the sensor
- Constants.SENS_OFF : turn off the sensor

And sensor unique id constants are:

- Constants.SENS_SMART_LDR : luminosity sensor
- Constants.SENS_SMART_FLOW_5V : flow sensor at 5V socket
- Constants.SENS_SMART_LCELLS_5V : load cell
- Constants.SENS_SMART_LCELLS_10V : load cell
- Constants.SENS_SMART_CURRENT : current clamp
- Constants.SENS_SMART_TEMPERATURE : temperature sensor
- Constants.SENS_SMART_HUMIDITY : humidity sensor
- Constants.SENS_SMART_US_3V3 : ultrasound sensor at 3.3V socket
- Constants.SENS_SMART_FLOW_3V3 : flow sensor at 3.3V socket
- Constants.SENS_SMART_US_5V : ultrasound sensor at 5V socket
- Constants.SENS_SMART_DFS_3V3 : foil sensor at 3.3V socket
- Constants.SENS_SMART_DFS_5V : foil sensor at 5V socket
3. Call the read method and store the value returned:

```java
long result = sm.readValue(int sensor);
```

A simple complete example of the use of this library for reading the current sensor:

```java
static SmartMetering sm = new SmartMetering();
sm.ON();
sm.setSensorMode(Constants.SENS_ON, Constants.SENS_SMART_CURRENT);
long current = sm.readValue(Constants.SENS_SMART_CURRENT);
Logger appendString(csr.s2b("Current measurement: "));
Logger.appendHexLong(Float.toLong(current, (byte)2));
Logger.flush(Mote.INFO);
```

Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.
See more examples at:


5.2.4. Agriculture Library

The `com.libelium.agriculture` is the responsible of managing the use of the sensor board called Agriculture.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section. To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```bash
> cd /API path/com/libelium/agriculture
> mrc --assembly=agriculture-1.0 --system=waspmote --ref=../common/common-1.0 Agriculture.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```bash
> cd/API path/com/libelium/agriculture
> moma-load agriculture
```

Here are the necessary steps to read a value from a sensor:

1. Create the object of the Agriculture library class and switch on the same:

   ```java
   static Agriculture agr = new Agriculture();
   agr.ON();
   ```

2. Set the state of the sensor that is gonna be read:

   ```java
   agr.setSensorMode(byte mode, int sensor);
   ```
Where mode constants are:
- Constants.SENS_ON : turn on the sensor
- Constants.SENS_OFF : turn off the sensor

And sensor unique id constants are:
- Constants.SENS_AGR_PRESSURE : Atmospheric Pressure Sensor
- Constants.SENS_AGR_WATERMARK1 : Soil Moisture 1 Sensor
- Constants.SENS_AGR_WATERMARK2 : Soil Moisture 2 Sensor
- Constants.SENS_AGR_WATERMARK3 : Soil Moisture 3 Sensor
- Constants.SENS_AGR_ANEMOMETER : Weather Station Sensor
- Constants.SENS_AGR_VANE : Weather Station Sensor
- Constants.SENS_AGR_DENDROMETER : Trunk diameter Sensor
- Constants.SENS_AGR_PT100 : Soil temperature Sensor
- Constants.SENS_AGR_LEAF_WETNESS : Leaf wetness Sensor
- Constants.SENS_AGR_TEMPERATURE : Temperature Sensor
- Constants.SENS_AGR_HUMIDITY : Humidity Sensor
- Constants.SENS_AGR_RADIATION : Solar radiation Sensor
- Constants.SENS_AGR_SENSIRION : Humidity and temperature Sensor
- Constants.SENS_AGR_LDR : Luminosity Sensor

3. Call the read method and store the value returned:

    long result = agr.readValue(int sensor);

The sensors that can be read this way are:
- Constants.SENS_AGR_PRESSURE : Atmospheric Pressure Sensor
- Constants.SENS_AGR_WATERMARK1 : Soil Moisture 1 Sensor
- Constants.SENS_AGR_WATERMARK2 : Soil Moisture 2 Sensor
- Constants.SENS_AGR_WATERMARK3 : Soil Moisture 3 Sensor
- Constants.SENS_AGR_ANEMOMETER : Weather Station Sensor
- Constants.SENS_AGR_VANE : Weather Station Sensor
- Constants.SENS_AGR_LEAF_WETNESS : Leaf wetness Sensor
- Constants.SENS_AGR_TEMPERATURE : Temperature Sensor
- Constants.SENS_AGR_HUMIDITY : Humidity Sensor
- Constants.SENS_AGR_LDR : Luminosity Sensor

A simple complete example of the use of this library for reading the temperature sensor:

    static Agriculture agr = new Agriculture();
    agr.ON();
    agr.setSensorMode(Constants.SENS_ON, Constants.SENS_AGR_TEMPERATURE);
    long temperature = agr.readValue(Constants.SENS_AGR_TEMPERATURE);
    Logger appendString(csr.s2b("Temperature measurement: ")));
    Logger appendHexLong(Float.toLong(temperature, (byte)2));
    Logger.flush(Mote.INFO);

To read the rest sensors there are unique methods that needs an special parameter that will be the invoked callback when the reading is done. This is because they are doing an asynchronous reading.

The callback method will receive a byte array containing the long result.

    public void readPT1000(Callback cb);
    public void readDendrometer(Callback cb);
    public void readRadiation(Callback cb);

    /*
     * Sensirion type:
     *    Constants.SENSIRION_TEMP
     *    Constants.SENSIRION_HUM
     */
    public void readSensirion(byte type, Callback cb);

    public void readSensirion(byte type, Callback cb);

And an example of the use of this library for reading one of the special sensor:

```java
static Agriculture agr = new Agriculture();
agr.ON();
agr.setSensorMode(Constants.SENS_ON, Constants.SENS_AGR_DENDROMETER);

agr.readDendrometer(new Callback(null) {
    public int invoke(byte[] buf, int len) {
        return PruebasAgricultura.callbackDendrometer(buf, len);
    }
});

private static int callbackDendrometer(byte[] buf, int len){
    Logger.appendString(csr.s2b("Medicion dendro: ");
    long result = Util.get32(buf, 0);
    Logger.appendHexLong(Float.toLong(result, (byte)2));
    Logger.flush(Mote.INFO);
    return 0;
}
```

This Library can also attach Pluviometer interrupts:

```java
agr.attachPluvioInt();
```

And then, the value of the actual count of pluviometer interruptions can be read as:

```java
int count = readPluviometerValue();
```

Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:

http://www.libelium.com/development/waspmote/examples/?cat=mr-sensor-boards&subcat=mr-agriculture

---

### 5.2.5. Events Library

The `com.libelium.events` is the responsible of managing the use of the sensor board called Smart Cities.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```bash
> cd /API path/com/libelium/events
> mrc --assembly=events-1.0 --system=waspmote --ref=../common/common-1.0 Events.java
```
To upload the compiled assembly to the Waspmote, run the following commands:

```
> cd/API  path/com/libelium/events
> moma-load  events
```

Here are the necessary steps to read a value from a sensor:

1. Create the object of the Events library class and switch on the same:
   ```java
   static Events events = new Events();
   events.ON();
   ```

2. Call the read method and store the value returned:
   ```java
   long result = events.readValue(int sensor, byte type);
   ```
   or (if it's none of the types described below)
   ```java
   long result = events.readValue(int sensor);
   ```

Where sensor unique id constants are:

- Constants.SENS_SOCKET1
- Constants.SENS_SOCKET2
- Constants.SENS_SOCKET3
- Constants.SENS_SOCKET4
- Constants.SENS_SOCKET5
- Constants.SENS_SOCKET6
- Constants.SENS_SOCKET7
- Constants.SENS_SOCKET8

And the types are:

- Constants.SENS_RESISTIVE
- Constants.SENS_FS100
- Constants.SENS_FS200A
- Constants.SENS_FS400
- Constants.SENS_TEMPERATURE
- Constants.SENS_HUMIDITY

A simple complete example of the use of this library for reading the temperature sensor:

```java
static Events events = new Events();
sc.ON();
long temperature = events.readValue(Constants.SENS_SOCKET5, Constants.SENS_TEMPERATURE);
Logger appendString(csr.s2b("Temperature measurement: "));
Logger.appendHexLong(Float.toLong(temperature, (byte)2));
Logger.flush(Mote.INFO);
```

This library also provides a system of interrupts that may be caused when a sensor detects that the value read is higher than an indicated threshold.

The steps to configure that threshold are the following:

1. Set the threshold value of the sensor:
   ```java
   events.setThreshold(Constants.SENS_SOCKET6, Float.make(20, (byte)-1));
   ```

2. Attach the interrupts and set the callback function that will be called when the interrupt is fired:
   ```java
   events.attachInt(new Callback(null) {
       public int invoke(byte[] buf, int len) {
           return callbackInt(buf, len);
       }
   });
   ```
3. Then, in the `callbackInt` method, discriminate the sensor that caused the interruption:

```java
byte flags = events.loadInt();
if((flags & Constants.SENS_SOCKET1) > 0))
```

A simple complete example of the use of this library for setting a threshold in the socket1 sensor and capture the interruption caused:

```java
static {
    events.ON();
    events.setThreshold(Constants.SENS_SOCKET1, Float.make(20, (byte)-1));
    sc.attachInt(new Callback(null) {
        public int invoke(byte[] buf, int len) {
            return callbackInterrupt(buf, len);
        }
    });
}

private static int callbackInterrupts(byte[] buf, int len){
    byte flags = sc.loadInt();
    if((flags & Constants.SENS_SOCKET1)>0){
        Logger.appendString(csr.s2b("Socket 1 INTERRUPT"));
        Logger.flush(Mote.INFO);
    }
}
```

Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:


### 5.2.6. Gases Library

The `com.libelium.gases` is the responsible of managing the use of the sensor board called Smart Metering.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```
> cd /API path/com/libelium/gases
> mrc --assembly=gases-1.0 --system=waspmote --ref=./common/common-1.0 Gases.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```
> cd/API path/com/libelium/gases
> moma-load gases
```
Here are the necessary steps to read a value from a sensor:

1. **Create the object of the Gases library class and switch on the same:**
   ```java
   static Gases gas = new Gases();
   sm.ON();
   ```

2. **Set the state of the sensor that is gonna be read:**
   ```java
   sm.setSensorMode(byte mode, int sensor);
   ```

Where mode constants are:
- `Constants.SENS_ON` : turn on the sensor
- `Constants.SENS_OFF` : turn off the sensor

And sensor unique id constants are:
- `SENS_PRESSURE` : Atmospheric Pressure Sensor
- `SENS_CO2` : Carbon Dioxide Sensor
- `SENS_O2` : Oxygen Sensor
- `SENS_SOCKET2A` : Sensor placed on SOCKET2A
- `SENS_SOCKET2B` : Sensor placed on SOCKET2B
- `SENS_SOCKET3A` : Sensor placed on SOCKET3A
- `SENS_SOCKET3B` : Sensor placed on SOCKET3B
- `SENS_SOCKET4A` : Sensor placed on SOCKET4

3. **Configure the sensor:**
   ```java
   public byte configureSensor(int sensor, byte gain, long resistor)
   ```

4. **Call the read method and store the value returned:**
   ```java
   long result = gas.readValue(int sensor);
   ```

A simple complete example of the use of this library for reading the current sensor:

```java
static Gases gas = new Gases();
gas.ON();
gas.setSensorMode(Constants.SENS_ON, Constants.SENS_SOCKET4CO);
gas.configureSensor(Constants.SENS_SOCKET4CO, (byte)2, Float.make(100, (byte)0));
long socket4CO = gas.readValue(Constants.SENS_SOCKET4CO);
Logger.appendString(csr.s2b("4CO measurement: "));
Logger.appendHexLong(Float.toLong(socket4CO, (byte)2));
Logger.flush(Mote.INFO);
```

Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:

5.2.7. Parking Library

The com.libelium.parking is the responsible of managing the use of the sensor board called Smart Metering.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```
> cd /API path/com/libelium/parking
> mrc --assembly=parking-1.0 --system=waspmote --ref=../common/common-1.0 Parking.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```
> cd/API path/com/libelium/parking
> moma-load parking
```

A complete example of reading the state of a parking place using this library:

```java
static{
    long coefX2 = Float.make(-1250745, (byte) -11);
    long coefX = Float.make(-1955917, (byte) -9);
    long constX = Float.make(1043527, (byte) -6);

    long coefY2 = Float.make(1434863, (byte) -12);
    long coefY = Float.make(-3030432, (byte) -10);
    long constY = Float.make(1006304, (byte) -6);

    long coefZ2 = Float.make(-7289752, (byte) -12);
    long coefZ = Float.make(6671079, (byte) -11);
    long constZ = Float.make(1001546, (byte) -6);

    park.loadReference(coefX2, coefX, constX, coefY2, coefY, constY, coefZ2, coefZ, constZ);
    park.readParkingSetReset();
    int temperature = park.readTemperature();
    park.calculateReference(temperature);
    boolean status = park.estimateState();

    if(status==Constants.PARKING_OCCUPIED){
        Logger.appendString(csr.s2b(“OCCUPIED”));
        Logger.flush(Mote.INFO);
    }else{
        Logger.appendString(csr.s2b(“EMPTY”));
        Logger.flush(Mote.INFO);
    }
}
```
Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:
http://www.libelium.com/development/waspmote/examples/?cat=mr-sensor-boards&subcat=mr-parking

5.2.8. Radiation Library

The `com.libelium.radiation` is the responsible of managing the use of the sensor board called Smart Metering.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```
> cd /API path/com/libelium/radiation
> mrc --assembly=radiation-1.0 --system=waspmote --ref=../common/common-1.0 Radiation.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```
> cd/API path/com/libelium/radiation
> moma-load radiation
```

This library offers a method to obtain the count of pulses of radiation in an interval.

The callback is a method that will be called when the `<timeMillisRadiation>` time has elapsed:

`getRadiation(byte mode, Callback cb, long timeMillisRadiation)`

A complete example of the use of this library to get the radiation count in ten seconds:

```java
static{
    rad.ON();
    rad.getRadiation((byte)0, new Callback(null) {
        public int invoke(byte[] buf, int len) {
            return interrupt(buf, len);
        }
    }, 10000);
}
static int interrupt(byte[] buf, int len) {
    long result = Util.get32(buf, 0);
    Logger.appendString(csr.s2b("Radiacion en usv/h: "));
    Logger.appendHexLong(Float.toLong(result,(byte)2));
    Logger.flush(Mote.INFO);
}
```
Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:


### 5.2.9. Prototyping Library

The `com.libelium.prototyping` is the responsible of managing the use of the prototyping sensor board.

**Note:** The necessary documentation to make the correct assembly of the sensors into the board and to know where must they be connected can be found in the technical guide of the board:


Please note that this guide is made for the original Waspmote system and the code that there appears will not work on Mote Runner.

For proper use of this library, the common assembly is needed to be loaded into the Waspmote. See “Common Library” section.

To build the library, this set of commands must be executed in the Mote Runner shell (mrsh):

```
> cd /API/path/com/libelium/prototyping
> mrc --assembly=prototyping-1.0 --system=waspmote --ref=./common/common-1.0 Prototyping.java
```

To upload the compiled assembly to the Waspmote, run the following commands:

```
> cd/API/path/com/libelium/prototyping
> moma-load prototyping
```

Here are the necessary steps to read a value from a sensor:

1. Create the object of the Prototyping library class and switch on the same:
   ```java
   static Prototyping proto = new Prototyping();
   proto.ON();
   ```

2. Call the read method and store the value returned:
   ```java
   long result = proto.readAnalogSensor(byte pin);
   ```

To read a value from the ADC that the board has attached:

1. Create the object of the Prototyping library class and switch on the same:
   ```java
   static Prototyping proto = new Prototyping();
   proto.ON();
   ```

2. Call the reading method providing the callback that should be used when the read is done:
   ```java
   readADC(new Callback(null){
   public int invoke(byte[] buf, int len) {
       return <Class>.callback(buf,len);
   }
});
```
A simple complete example of the use of this library for reading a sensor:

```java
static Prototyping proto = new Prototyping();
proto.ON();
long valRead = proto.readAnalogSensor(byte pin);

Logger appendString(csr.s2b("Sensor measurement: "));
Logger appendHexLong(Float.toLong(valRead, (byte)2));
Logger.flush(Mote.INFO);
```

Further documentation of the methods can be found embedded within the code and in a generated javadoc delivered with the libraries.

See more examples at:

http://www.libelium.com/development/waspmote/examples/?cat=mr-sensor-boards&subcat=mr-prototyping
6. Mote Runner SDK

The latest Mote Runner SDK for Windows, Linux (and Linux 64-bit) and Mac iOS can be downloaded from:
http://www.zurich.ibm.com/moterunner

6.1. SDK Installation

6.1.1. Requirements

6.1.1.1. Firefox

The Firefox browser is required to operate the IBM Mote Runner Web GUI and for reading the documentation. These features will not work correctly under Internet Explorer or other browsers. You should have at least version 23.x installed. You can download Firefox [here].

6.1.1.2. Java JRE/SDK

You need a Java JRE to operate the IBM Mote Runner tool chain. If you are planning to write IBM Mote Runner applications in Java then you need a Java JDK containing a Java compiler. You should have installed a Java JRE/JDK version 6 or higher (download [here]). If in doubt install the JDK.

Make sure the tools directory of the JDK/JRE is added to the PATH environment variable so that the IBM Mote Runner compiler can find the java and javac programs.

6.1.1.3. Optional: C# Language Compiler

You only need a C# compiler if you plan to develop IBM Mote Runner applications in the C# programming language. Mono is available [here].

6.1.2. Windows

6.1.2.1. Quick Start

- Download and unzip the corresponding moterunner-* .win32.zip file
- Execute the setup .hta file and follow the instructions
- Set-up the PATH environment variable

6.1.2.2. PATH Environment Variable

The PATH environment variable should contain all the directories holding tools relevant for IBM Mote Runner. In addition, you should add the IBM Mote Runner tools directory <YOUR_INSTALLATION_DIRECTORY>/moterunner/win32/bin to the PATH.

For example assuming you have installed IBM Mote Runner under C:\moterunner:

PATH=C:\moterunner\win32\bin;%PATH%

You can double click on the file mrcmd.bat in the moterunner directory to get a Windows command shell with the path set.
6.1.2.3. Optional: C# development

Nowadays, all Windows operating systems have the .NET framework installed by default. Any version (v1, v2, or v3) will do. You can explicitly select the version with an option for the IBM Mote Runner compiler. Verify that the framework is installed in the default place (under C:/WINDOWS/Microsoft.NET/Framework). If it is not in this location make sure that the directory is listed in the PATH environment variable so that the csc.exe and other tools can be found by the IBM Mote Runner compiler. You can also use Mono on Windows (see above).

6.1.2.4. Optional: Cygwin

Furthermore, you might consider installing Cygwin. Although not mandatory, it might make running some sample code out of the box easier. Cygwin emulates a Unix like environment on Windows. Some examples and sample code use small makefiles and tool usage throughout the documentation assumes a Unix like tool chain (GNU make, bash shell etc.).

Sample code and makefiles are kept very simple and can be easily changed into your preferred tool chain with little effort. If desired, Cygwin can be downloaded here.

6.1.3. Linux / Linux (64-bit)

6.1.3.1. Quick Start

- Download and unzip the corresponding moterunner-*linux*.zip file
- Execute the setup script and follow the instructions
- Move the moterunner directory to a location of your own choice, e.g. ~/moteruner
- Set-up the PATH and LD_LIBRARY_PATH environment variables

6.1.3.2. PATH and LD_LIBRARY_PATH Environment Variable

The PATH environment variable should contain all the directories holding tools relevant for IBM Mote Runner. In addition, you should add the IBM Mote Runner tools to the PATH environment variable.

For example assuming you have installed IBM Mote Runner under ~/moteruner you can execute the following commands in your shell:

Linux (32-bit)

    export PATH=~/moterunner/linux/bin:$PATH
    export LD_LIBRARY_PATH=~/moterunner/linux/bin:$PATH

Linux (64-bit)

    export PATH=~/moterunner/linux64/bin:$PATH
    export LD_LIBRARY_PATH=~/moterunner/linux64/bin:$PATH

6.1.3.3. Optional: C# development

Mono is available at Mono Project.

6.1.4. MAC OSX

6.1.4.1. Quick Start

Download and unzip the corresponding moterunner-*macosx*.zip file

Install the dmg file and follow the onscreen instructions

Set-up the PATH and LD_LIBRARY_PATH environment variables
6.1.4.2. PATH Environment Variable

The PATH environment variable should contain all the directories holding tools relevant for IBM Mote Runner. In addition, you should add the IBM Mote Runner tools directory to the PATH environment variable.

For example, assuming you have installed IBM Mote Runner under ~/moterunner:

```bash
export PATH="~/moterunner/macosx/bin:$PATH"
```

6.1.4.3. Optional: C# development

Mono is available [here](#).

6.1.5. Optional: Eclipse Integration

The Mote Runner compiler tool can be integrated with the Eclipse SDK for Java.

See complete Eclipse integration at:


Note that mrsh should be running in order to access the mentioned link

6.2. Firmware Installation for Waspmote

In order to use Mote Runner with the Waspmote the Mote Runner firmware must be installed on the Waspmote hardware. The following hardware and software are required:

- A Waspmote PRO from Libelium with: a mini USB cable, an extension RF23x radio board, an extension Ethernet board
- An ISP (or JTAG) programmer for AVR from Atmel
- Flash writing tools like the Atmel Studio from Atmel or avrdude
- The Mote Runner firmware `<mote_runner_directory>/firmware/waspmote.hex`

6.2.1. Flashing the hex image

1. First of all connect the Waspmote to a power supply (for example to a computer) using the micro-USB cable. The Waspmote should be powered on (on the image move the switch to the left).
2. The next step is to connect the AVR programmer to the PC using the USB cable.

3. Now the AVR programmer can be connected to the ICSP connector on the back of the Waspmote.

4. The last step is to load the firmware on the waspmote using avrdude or Atmel Studio as explained in the following sections.
6.2.1.1. Loading the firmware using avrdude

Just type on a terminal the following commands.

Set the fuses:

```
sudo avrdude -c atmelice_isp -p m1281 -P usb -b 115200 -e -u -U efuse:w:0xFF:m -U hfuse:w:0xD0:m -U lfuse:w:0xFF:m
```

Load the firmware image:

```
sudo avrdude -c atmelice_isp -p m1281 -P usb -b 115200 -U flash:w:waspmote.hex
```

6.2.1.2. Loading the firmware using Atmel Studio

Start the Atmel Studio and connect to the programmer. To check that the programmer and the SPI interface is operating correctly, select the ATmega1281 as the device and read the voltage and signature.

![Atmel Studio AVR Programming Interface settings](image)

*Figure: Atmel Studio AVR Programming Interface settings*
Now, erase the entire chip. Set the fusses as described below. And finally, flash the Mote Runner firmware hex image.

6.3. Application development

Mote Runner applications can be developed using Java or C#.

Once an application has been written it must be compiled using the Mote Runner Compiler (mrc).

This tool can be found on the /bin folder inside the Mote Runner folder. Is useful to have this and other Mote Runner tools in the PATH environment variable.

The compiler generates an assembly. An assembly is bytecode that the VM inside the mote can execute.

6.3.1. Compiling applications

The basic Mote Runner compiler invocation is as follows:

```
mrc --assembly=<name>-.<major>-.<minor> --system=waspmote <code_file>
```

For example an application code on the file TestADC.java can be compiled as follows:

```
mrc --assembly=testadc-1.0 --system=waspmote TestADC.java
```
The compilation process will create two files: binary assembly (sba) and exchange file (sxp). The sxp file describes the public API of the assembly and it is needed in order to allow other assemblies to call its methods.

Public methods in Java applications and non private in case of C# will be visible by other assemblies on the mote and could be called by them.

In order to compile an application that calls other assembly methods the sxp file should be referenced:

```
mrc --assembly=<name>-<major>.<minor> --system=wasp mote -r:<ref_sba>-<major>.<minor> <code_file>
```

For example if we want to reference the assembly `temperature-1.0` from the application `TestADC.java` we should do:

```
mrc --assembly=testadc-1.0 --system=wasp mote -r:temperature-1.0 TestADC.java
```

**6.3.2. Major.Minor Version Numbers**

To facilitate application development and evolving APIs, the Mote Runner tool chain keeps manages binary compatibility of evolving assemblies. Java and C# runtime systems use strings to describe the signature of a method in detail. Changing the signature of a method will break linking. A Mote Runner platform uses small integer values for linking to save space. These integer values have a well defined meaning in the context of a specific assembly major version. The compiler manages the assignment if it identifies links across minor versions. If a programmer changes the public API of an assembly the compiler will ensure compatibility to the current or last minor version.

See complete Mote Runner compiler documentation at:


Note that mrsh should be running in order to access the mentioned link

**6.3.3. Uploading applications**

Communication between waspmotes (real ones or simulated) and the computer is established through Sonoran.

We can create a sonoran process running the Mote Runner Shell (mrsh).

**6.3.3.1. Uploading assemblies to simulated motes**

Inside mrsh:

```
# We start a saguaro process. We could specify a host and a port with saguaro-start -h localhost:44044 for example.
saguaro-start

# Now we get connected to the saguaro process. If there is more than one saguaro process we can specify to which one we want to connect by preceding the command with the saguaro-process ID.
saguaro-connect

# We create a mote on the process we are connected. With -d option we specify the platform
mote-create -d waspmote

# An assembly can be loaded on this simulated mote. For example we load the assembly test-1.0
moma-load test-1.0

# For removing that assembly from the mote.
moma-delete test-1.0
```
6.3.3.2. Uploading assemblies to real motes

Establishing connection with the mote through USB cable (Inside mrsh):

1. **If we know the port to which it is connected (e.g. /dev/ttyUSB0):**
   
   ```
   mote-create -p /dev/ttyUSB0
   ```

2. **If we don’t know it:**
   
   ```
   # With the waspmote unplugged:
   lip-enumerate -w –waspmote -t30s
   
   # Now we plug the waspmote to the computer.
   # We can see a list of mrsh environment variables available with
   # the list-vars command.
   list-vars
   
   # One of the variables should be WASPMOTE_MR_PORT that
   references the port to which the waspmote is connected.
   # In order to establish the connection we run:
   mote-create -p${WASPMOTE_MR_PORT}
   ```

Once a connection between the computer and the waspmote is established we can upload and remove assemblies.

```
# Loading the assembly test-1.0 on the mote.

moma-load test-1.0

# Removing test-1.0 from the mote.

moma-delete test-1.0
```

See complete assembly life cycle documentation at:

[http://localhost:5000/doc/system/index.html#system_rtenv_1](http://localhost:5000/doc/system/index.html#system_rtenv_1)

Note that mrsh should be running in order to access the mentioned link.
6.3.4. Debugging applications

When using the Mote Runner simulation applications can be debugged using an Eclipse plug in as shown in the image below.

![Mote Runner Debugger plugin for Eclipse](image)

Figure: Mote Runner Debugger plugin for Eclipse

See details about installing the plug-in here:
http://localhost:5000/doc/system/index.html#start_startFirstSteps_PluginInstallation

See details about using the plug-in here:
http://localhost:5000/doc/system/index.html#tools_mrdebugger_0

Note that mrsh should be running in order to access the mentioned links
6.3.5. Simulating applications

The Mote Runner SDK comes with a rich simulation environment to facilitate application development and debugging. The environment accurately simulates the memory access, radio communication, sensor interaction, LED activity, ADC readings, GPIO operations, custom I2C sensors, position, and power consumption.

In general, it is good practice to first develop applications using the simulation environment before moving to actual hardware.

The mrsh commands provided by the moma-, mote-, device-, and feed- groups allows controlling most of the properties of simulated motes. For advanced control of I2C devices scripting using the Sonoran Java script environment is available.

To create a simulated Waspmote execute the following command in the mrsh:

```bash
>mote-create -d dll waspmote
```

To list all available devices for the simulated mote can be listed:

```bash
>mote-info -devices
```

To turn on all LEDs of a Waspmote once can use the following command:

```bash
>moma-leds 3
```

To feed the simulation with data for the GPIO and ADC operations the feed-start and feed-stop commands can be used.

### 6.3.5.1. Timeline

All events (e.g., LED state changes, Radio messages) which occur in the simulation can be visualized using a Web-based Timeline ([http://localhost:5000/timeline](http://localhost:5000/timeline)) as shown in the image below.

![Figure: The web tool Timeline for Mote Runner](image-url)
6.3.6. Managing Applications MOMA

The MOMA (Mote Manager) protocol specifies a number of LIP messages to configure a Mote Runner system. Any wireless or wired network protocol implementation can tunnel LIP messages to the system to initiate these MOMA management functions. The following ‘mrsh’ commands are wrappers to a LIP based communication using ‘mrsh’ sockets.

- moma-wlip: configure the persistent settings for a wireless WLIP mote such as channel.
- moma-list: list assemblies on a mote.
- moma-load: load an assembly.
- moma-delete: delete an assembly on a mote.
- moma-factory-reset: reset the mote and bring it back to factory state, e.g. deleting all installed applications
- moma-leds: switch on/off LEDs on motes.

Use ‘help moma’ in ‘mrsh’ to see the list of moma commands and ‘help moma-xxx’ for a command specific help message.

6.4. Libraries

The standard libraries available on the Mote Runner platform are available in the global assembly cache (GAC) folder, which can be found in ~/moterunner/gac. The gac directory contains the following:

For each assembly version (name-major.minor the GAC holds the following types of files:

- a set of sba files: compiled binary versions of the assembly
- a set of sdb files: debugging information for each sba files (optional)
- sxp file: the API definition of the assembly
- jar file: the Java API of the assembly (optional)
- dll file: the C# API of the assembly (optional)

**Note:** to use any libraries with Eclipse and code completion, the corresponding .jar file would need to be added to your Java project built path.

See complete GAC documentation at:
http://localhost:5000/doc/system/index.html#tools_mrgac_0

Note that mrsh should be running in order to access the mentioned link

6.4.1. Saguaro System

Mote Runner system library which contains all hardware independent APIs such as for example the LIP, ADC, I2C, and Radio class as well as corresponding callbacks.

Note that the saguaro-system library is pre-loaded onto all simulated and hardware motes and cannot be deleted.

See Saguaro API documentation at:

Note that mrsh should be running in order to access the mentioned link
6.4.2. WaspMote System

The waspmote-system library contains specific constants for pins, ADC settings, and device identifiers for the WaspMote hardware. The library is pre-loaded onto all WaspMote simulated and hardware motes and cannot be deleted.

To use the waspmote-system library simply use the import statement in Java (or the using statement in C#).

Java: import com.ibm.waspmote.*;

C#: using com.ibm.waspmote;

In addition, when compiling the application you need to specify the reference to the library. An example compiler invocation is as follows:

```
  mrc -ref:waspmote-system-#.# example.java
```

See WaspMote System API documentation at:

Note that mrmsh should be running in order to access the mentioned link.

6.4.3. Logger

Mote Runner provides a simple logging facility for applications attached via LIP to a host or via WLIP to a WLIP gateway. This facility is conveniently exported to user applications by the ‘logger’ assembly. A user application can link against the logger assembly and use various static methods to create log messages and its ‘flush’ method to send the log messages to the host. The log messages are transported over WLIP and LIP, are picked up by ‘mrmsh’ and appended to the shell log or dumped on the console.

Code using the Logger may look like this:

```
  Logger.appendString(csr.s2b("value = "));
  Logger.appendInt((int) 0);
  Logger.flush(Mote.DEBUG);
```

This appends a string to the log buffer, appends a '0' and sends the log buffer contents to the host. The severity of the log message is 'DEBUG' (instead of INFO, WARN or ERROR). ‘csr.s2b’ is a helper method which is already executed when the code is compiled to a Mote Runner assembly: it maps the specified String to a byte array and adds that to the destination assembly.

The ‘log-conf’ command in the ‘mrmsh’ allows to specify the logging behavior dependent on the category and severity of a message. Logging messages from motes appear under the category ‘MAPP’. To get all log messages from a mote starting with severity DEBUG immediately debugged on the console, the following ‘mrmsh’ command may be used:

```
  log-conf -i MAPP DEBUG
```

Log messages are kept by ‘mrmsh’ in an internal buffer, the following prints the last thirty messages received by ‘mrmsh’:

```
  log-show -c 30
```

Note that log messages are discarded by the system when they are issued on a wireless mote running a custom network protocol.
6.4.4. WLIP Gateway

WLIP is a simple single channel star based wireless built-in network protocol. When a wireless mote starts it it sends out ‘HELLO’ messages listening for an acknowledgement by a gateway. If no WLIP gateway answers, an application can take over the radio and start a custom network protocol. If a WLIP gateway picks up the ‘HELLO’ it records the address of the mote and uses that for the future identification of and communication with the mote. Once connected, LIP messages can be exchanged between WLIP gateway and mote and especially the ‘moma’ commands in the ‘mrsh’ are applicable to manage wireless motes and upload new applications.

The WLIP gateway functionality is not built-in but provided by the WLIP gateway assembly. It is downloaded using the `moma-load` command in the ‘mrsh’. It is configured and managed using LIP messages where a number of ‘mrsh’ commands simplify the handling of WLIP.

The following command uploads the most recent version of the assembly on the simulated mote ‘a0’ and starts the WLIP gateway protocol on it:

```
a0 wlip-setup
```

Use ‘help wlip-setup’ to see the options for the command, e.g. to modify the channel used by the WLIP gateway.

```
a0 wlip-appeal
```

A ‘wlip-appeal’ lets the WLIP gateway request ‘HELLO’ messages from the wireless motes and is useful to gather a list of wireless motes still connected to the gateway.

WLIP is not a battery efficient protocol as wireless motes never switch off the receiver. Messages are sent using CSMA/CA and are retransmitted on error for a configurable number of times. The destination mote for a unicast message is addresses using its EUI-64 (extended address). Broadcast messages are received by all WLIP motes in range.

6.4.5. 6LoWPAN / MRv6

Documentation about 6LoWPAN library can be found at section “6LoWPAN Network”
6.5. Examples and demos

Mote Runner comes with a large number of examples and demo applications, including using the generic Mote Runner API to implement sensor drivers, as well as controlling the radio for custom wireless protocols. The examples folder from the distribution contains the source code and further documentation for each example.

A good starting point with Mote Runner examples are the interactive tutorials. To access the interactive tutorials, simply start the Mote Runner shell (mrsh) process, then open Firefox, and go to http://localhost:5000. Access the Demos category, as highlighted in the picture below.
The tutorial uses the Web-based Mote Runner shell shown below. The Web mrsh can be accessed at http://localhost:5000/mrshell. Compared to the standard command-line interface, it brings a number of additional features such as an LED widget. The LEDs widget which can be loaded and activated using the mrshell-widget leds command.

The Web-based shell is extensible and additional widgets can be created using HTML and Javascript.

### 6.6. Mote Runner API Documentation

More details about the architecture can be found in the online documentation (http://localhost:5000/doc).

Note that the Mote Runner Shell (mrsh) must be started to access the online documentation. Also note that the supported Web browser is Firefox.

---

**Figure:** Mote Runner documentation entry point
7. 6LoWPAN / MRv6

7.1. Introduction

Mote Runner comes with a 6LoWPAN-based protocol, called MRv6, which can be installed on motes to provide IPv6-based connectivity.

MRv6 is a TDMA based multi-hop network which allows for an IPv6 based communication between hosts in the Internet and motes in the wireless network.

Datagram packets transferred in the wireless network make use of the IP header compression (IPHC) as specified in RFC4944 and RFC6282 for 6LoWPAN [1] [2]. In contrast to LIP and WLIP, MRv6 is not a built-in component, but fully implemented in C#, packaged in assemblies and installed using the default Mote Runner upload mechanisms. The protocol is suited for data gathering applications where motes periodically send data to a remote host and are rarely subject to updates or requests from remote hosts. Communication between motes is expected to be rather uncommon, too.

The whole MRv6 network tree is managed by the edge mote and only very limited routing functionality is implemented in the wireless motes. The edge mote decides upon association requests, assigns communication schedules between wireless nodes, and determines the routes in the network. A host application can freely detach from or attach to an established network without interfering with the tree management. Once attached, it can query the network, communicate with motes and retrieve data from them.

The format of application data packets exchanged in the network adheres to a subset of the 6LoWPAN specifications and constitutes a compressed representation of standard IPv6 packets. MRv6 features a tunnel program on OSX and Linux for hosts attached to an edge mote which maps IPv6 to 6lowpan packets and vice versa and forwards them between remote Internet hosts and the edge mote. As the tunnel implements a virtual network interface, the mapping occurs seamless for host and mote applications. Both wireless motes and hosts can be reached and identified using IPv6 addresses.

MRv6 supports a shell and programming interface to manage, debug, query and communicate with a wireless network, either physical or simulated. The tunnel is only required if communication with external host applications based on IPv6 is necessary.

7.2. 6LoWPAN Libraries

The 6LoWPAN library for mote runner is called MRv6.

In contrast to the default communication protocols LIP and WLIP, MRv6 is not a built-in Mote Runner component, but fully implemented in C#, packaged in assemblies and installed using the default Mote Runner upload mechanisms. The protocol is suited for data gathering applications where motes periodically send data to a remote host and are rarely subject to updates or requests from remote hosts. Communication between motes is expected to be rather uncommon, too.

The whole MRv6 network tree is managed by the edge mote and only very limited routing functionality is implemented in the wireless motes. The edge mote decides upon association requests, assigns communication schedules between wireless nodes, and determines the routes in the network. A host application can freely detach from or attach to an established network without interfering with the tree management. Once attached, it can query the network, communicate with motes and retrieve data from them.

The format of application data packets exchanged in the network adheres to a subset of the 6lowpan specifications and constitutes a compressed representation of standard IPv6 packets. MRv6 features a tunnel program for hosts attached to an edge mote which maps IPv6 to 6lowpan packets and vice versa and forwards them between remote Internet hosts and the edge mote. As the tunnel implements a virtual network interface, the mapping occurs seamless for host and mote applications. Both wireless motes and hosts can be reached and identified using IPv6 addresses.

MRv6 supports a shell and programming interface to manage, debug, query and communicate with a wireless network, either physical or simulated. The tunnel is only required if communication with external host applications based on IPv6 is necessary.
Files:

(Located on the <Mote runner folder>/examples/mrv6)

- **tunnel**: a simple program to setup a virtual IPv6 network interface which routes packets between the network interface and a hardware edge mote or between the network interface and Sonoran (and the simulation). Only supported on OSX and linux.
- **src**: directory with sources of the MRv6 protocol implementation, for both edge as well as wireless motes.
- **apps**: folder with sample applications.
- **java**: a library to be executed on a host. The library connects to the edge mote using UDP and allows for the event notification about mote appearances and disappearances. It is installed in lib/java.
- **lib**: contains the folder ‘js’ with the Javascript code for Sonoran to support the network protocol in the simulation and the compiled java library.

More examples can be found in examples/mrv6 inside the Mote Runner folder.

Complete API documentation can be found after generating it here:


In order to compile and generate MRv6 libraries documentation run the following commands on examples/mrv6/src:

```sh
make clean; make
doc
```

### 7.3. Using MRv6 Libraries

On this example one mote will be configured as the edge-mote (gateway) and some other motes will be end-devices. Using the Linux or Mac OSX tunnel application a virtual ipv6 interface will be configured in order to route information between the edge mote and other ipv6 networks. Nodes of this WSN will be accessible using its IPv6 address (through the edge-mote that is connected to the tunnel application).

#### 7.3.1. Setup

First of all MRv6 libraries must be compiled before using them.

In order to compile and generate MRv6 libraries documentation run the following commands on examples/mrv6/src folder:

```sh
make clean; make
doc
```

1. Connect one Wasp mote to a PC. Open *mrsh* on that PC and load the assembly *mrv6-edge* on the Wasp mote. This assembly is located on the gac so there is no need to specify the path. This Wasp mote will be the edge-mote. Once the assembly is loaded we can disconnect the Wasp mote from the PC.

Instructions to load the assembly using USB cable (from mrsh):

```sh
> mote-create -p /dev/ttyUSB0
> moma-load mrv6-edge
```
This mote will be connected through an Ethernet interface. The network configuration can be set using `moma-ipv4` command. This configuration will be stored on the EEPROM memory so it will stay configured even if the mote is restarted. For example network parameters can be set as follows (inside mrsh):

```bash
> moma-ipv4 --ip 192.168.1.226/24 --gateway 192.168.1.1 --udp 9999
```

2. Load the `mrv6-lib` assembly on the end nodes. Connect them to a PC, this step can be applied one by one to every mote. This assembly is located on the gac so there is no need to specify the path.

Instructions to load the assembly using USB cable (from mrsh):

```bash
> mote-create -p /dev/ttyUSB0
> moma-load mrv6-lib
```

3. Load an application assembly that uses MRv6 libraries on every end node. For this example the application used will be reply located in `examples/mrv6/apps/reply` inside the Mote Runner folder. For compiling the application the following commands should be executed inside the `examples/mrv6/apps/reply` directory:

```bash
> make clean; make
```

Once the application is compiled we can load it on the motes. From mrsh inside `examples/mrv6/apps/reply` directory (this may be repeated with every mote):

```bash
> mote-create -p /dev/ttyUSB0
> moma-load sense
```

4. Launch the tunnel application on the PC

### 7.3.2. Running the example

1. Now all motes should be unplugged from the PC, USB connector is no longer needed. Radio modules and USB connector can't work at the same time, so battery should be used. [NOTE: you can have the battery recharged the whole time by plugging the USB cable to a power plug socket by using the 110/220V to 5V transformer provided by Libelium. See section “USB connection and Radio Modules” for more information]. The edge-mote is the only one that should be connected through Ethernet Module directly to the PC (using a crossover Ethernet cable) or to a network accessible by the PC (using Ethernet cable). This edge-mote should be accessible from the PC if network configuration on the mote (done in step 1) is suitable for your PC or network configuration. Lights on the Ethernet Module should blink when it is connected to the network. We can test if the mote is accessible doing a ping to its ipv4 address.

2. The end nodes must be powered only by battery (or USB to power plug) and the 6LoWPAN module should be attached to the mote.

3. In order to start the edge mote run the following commands in mrsh:

```bash
> source ../../../lib/js/mrv6.js

> a0 v6-setup --MAX_DEPTH=12 --NUM_CHILDREN=5 --MAX_CHILDREN=5 --MAX_MOTES=6 --RECV_SAFETY_MILLIS=3 --R24_SLOT_RCV_MILLIS=12 --R24_SLOT_GAP_MILLIS=15 --R24_BEACON_GAP_MILLIS=20 --INFO_INTERVAL_CNT=0
```

4. As end-nodes are switched on they attach automatically to the WSN network so the edge-mote is now aware of new nodes. Different routes can be established automatically, maybe some nodes reach directly to the edge-node or maybe they are too far and they use other end-nodes to reach the edge-node.

5. In this example UDP packets can be sent to a mote using its IPv6 address and the mote send back the same UDP packet.

IPv6 addresses from nodes can be retrieved for example with the command `v6-connect` in mrsh:

```bash
> v6-connect
```

For example a node can have the EUI-64 IP address: `0200:0000:00AE:2F01`
Due to the tunnel application default configuration nodes can be reached using \texttt{fc00:db8:5::<EUI-64>}, so for this example the node that is going to be used will be accessible with the IP \texttt{fc00:db8:5::0200:0000:00AE:2F01}.

Information can be sent to the nodes from a machine capable to reach the PC that is running the tunnel application using IPv6. For simplicity it can be done from the same PC running the tunnel application. The information in this example will be sent using the Netcat tool (IPv6 version). Using it for sending UDP packets is as follows:

\begin{verbatim}
> nc6 -u fc00:db8:5::0200:0000:00AE:2F01 102
\end{verbatim}

Now for example the phrase “hello world” can be sent just typing it and pressing enter.

This should return the same phrase:

\begin{verbatim}
> nc6 -u fc00:db8:5::0200:0000:00AE:2F01 1024
   hello world
   hello world
\end{verbatim}

\section*{7.4. Quick start development with the Waspmote Mote Runner Lab Kit}

On section “Using MRv6 libraries” is explained how to configure WSN, how to compile MRv6 libraries and how to load applications using those libraries.

In this section Lab Kit motes will be running a custom application based on the reply application (<Mote Runner folder>/examples/mrv6/apps/reply).

This application will send back an analog reading from ADC channel 0 instead of replying the same information that it receives.

The value returned by the ADC will be converted by the application to ASCII characters representing the hexadecimal value of channel 0 lecture.

Information will be sent from a client using Netcat and it will reach the mote specified by its IPv6 address, the mote will read ADC 0 and send the value back to the client.
7.4.1. Application code

The application code is the following

adcread.cs:

```csharp
namespace com.ibm.saguaro.gateway.generic {
    using com.ibm.saguaro.system;
    using com.ibm.saguaro.mrv6;
    using com.ibm.saguaro.util;

    public class ReplySocket : UDPSocket {
        internal static uint LOCAL_PORT = 1024;
        internal static ReplySocket socket = new ReplySocket();
        internal static ADC adc;

        public ReplySocket() {
            this.bind(LOCAL_PORT);
            adc = new ADC();
            adc.open(0x01,GPIO.NO_PIN,0, 0);
        }

        public override int onPacket(Packet packet) {
            // An UDP packet has reached this mote
            LED.setState(2, (byte)(LED.getState(2)^1));

            // We read ADC channel 0
            uint adcRead = adc.readChannel(0);

            // We store the ADC reading on a byte array
            byte[] buf = new byte[2];
            Util.set16be(buf,0,adcRead);

            // ADC value hexadecimal representation is transformed to ASCII.
            // Doing so we can see the hexa value directly on Netcat
            buf = adc2Ascii(buf);

            uint len = packet.payloadLen;
            try {
                packet.swap(len);
            } catch {
                return 0;
            }

            Util.copyData(buf, 0, packet.payloadBuf, packet.payloadOff, 4);
            this.send(packet);
            return 0;
        }

        // Get ASCII representation of a value in hexadecimal
        private byte[] adc2Ascii(byte[] adcValue){
            byte[] aux = new byte[4];
            aux[0] = (byte)((adcValue[0] & 0xF0) >> 4);
            aux[1] = (byte)(adcValue[0] & 0x0F);
            aux[2] = (byte)((adcValue[1] & 0xF0) >> 4);
            aux[3] = (byte)(adcValue[1] & 0x0F);

            for(int i=0;i<4;i++){
                if(aux[i] < 9){
```
This application must be compiled referencing the MRv6 library mrv6-lib:

```
> mrc -assembly=adcread-1.0 -ref=mrv6-lib adcread.cs
```

At this point the assembly adcread-1.0 has been created and it should be uploaded to WSN nodes.

### 7.4.2. Retrieving values from the WSN

Once the network is configured, tunnel is running and the application is uploaded to the end-nodes they can be queried to send back the ADC 0 channel reading.

IPv6 addresses from nodes can be retrieved for example with the command v6-connect in mrsh:

```
> v6-connect
```

For example a node can have the EUI-64 IP address: 0200:0000:00AE:2F01

Due to the tunnel application default configuration nodes can be reached using fc00:db8:5::<EUI-64> so for this example the node that is going to be used will be accessible with the IP fc00:db8:5:0200:0000:00AE:2F01

Information can be sent to the nodes from a machine capable to reach the PC that is running the tunnel application using IPv6. For simplicity it can be done from the same PC running the tunnel application. The information in this example will be sent using the Netcat tool (IPv6 version). Using it for sending UDP packets is as follows:

```
> nc6 -u fc00:db8:5:0200:0000:00AE:2F01 102
```

Now 4 bytes need to be sent to the node. For example "0000". Typing 0000 and pressing enter will send those bytes to the node.

The value from the ADC will appear in screen:

```
> nc6 -u fc00:db8:5:0200:0000:00AE:2F01 1024
0000
03FF
```

In this case the node sent 03FF (1023 in decimal representation) which is the maximum value the ADC can return. With the default threshold configured on the ADC this means 3.3V.

Four bytes are sent to the node because the application previously loaded on the node reuse the same packet structure for sending the value back, and the ASCII representation of the ADC value needs four bytes.
8. Support

There is a specific thread in the Forum which will be supported by both Libelium and IBM teams.

- Libelium Dev Team will answer to the Hardware questions.
- IBM Dev Team will answer to the Software / Networking / Data related questions.

9. Documentation Changelog

From v4.0 to v4.1:

• Changed Weather Meters name to Weather Station WS-3000

From v4.1 to v4.2:

• Modifications in 2 console commands because the AVR programmer model changed
• Changed the batteries from 2300 mAh to the high-capacity 6600 mAh
10. Certifications

10.1. CE

In accordance with the 1999/05/CE directive, Libelium Comunicaciones Distribuidas S.L. declares that the Waspmote device conforms to the following regulations:

EN 55022:1998
EN 61000-4-3:2002
EN 61000-4-3/A1:2002
EN 61000-4-3:2006
UNE-EN 60950-1:2007

Compliant with ETSI EN 301 489-1 V1.6.1, EN 300 328, Date: March 26, 2009

If desired, the Declaration of Conformity document can be requested using the Contact section at:

http://www.libelium.com/contact

Waspmote is a piece of equipment defined as a wireless sensor capture, geolocation and communication device which allows:

• short and long distance data, voice and image communication
• capture of analog and digital sensor data directly connected or through probes
• wireless access enablement to electronic communication networks as well as local networks allowing cable free connection between computers and/or terminals or peripheral devices
• geospatial position information
• interconnection of wired networks with wireless networks of different frequencies
• interconnection of wireless networks of different frequencies between each other
• output of information obtained in wireless sensor networks
• use as a data storage station
• capture of environmental information through interface interconnection, peripherals and sensors
• interaction with the environment through the activation and deactivation of electronic mechanisms (both analog and digital)
10.2. FCC

Waspmote models:

Model 1- FCC (XBee PRO series 1 OEM + SIM900 GSM/GPRS module)
**FCC ID: XKM-WASP01** comprising
- FCC ID: OUR-XBEEPRO
- FCC ID: UDV-0912142009007

Model 2- FCC (XBee PRO ZB series 2 + SIM900 GSM/GPRS module)
**FCC ID: XKM-WASP02** comprising
- FCC ID: MCQ-XBEEPRO2*
- FCC ID: UDV-0912142009007

Model 3 - FCC (XBee 900MHz + SIM900 GSM/GPRS module)
**FCC ID: XKM-WASP03** comprising
- FCC ID: MCQ-XBEE09P
- FCC ID: UDV-0912142009007

**Installation and operation of any Waspmote model must assure a separation distance of 20 cm from all persons, to comply with RF exposure restrictions.**

**Module Grant Restrictions**

**FCC ID OUR-XBEEPRO**

The antenna(s) used for this transmitter must be installed to provide the separation distances, as described in this filing, and must not be co-located or operating in conjunction with any other antenna or transmitter. Grantee must coordinate with OEM integrators to ensure the end-users of products operating with this module are provided with operating instructions and installation requirements to satisfy RF exposure compliance. Separate approval is required for all other operating configurations, including portable configurations with respect to 2.1093 and different antenna configurations. Power listed is continuously variable from the value listed in this entry to 0.0095W

**FCC ID MCQ-XBEEPRO2**

OEM integrators and End-Users must be provided with transmitter operation conditions for satisfying RF exposure compliance. The instruction manual furnished with the intentional radiator shall contain language in the installation instructions informing the operator and the installer of this responsibility. This grant is valid only when the device is sold to OEM integrators and the OEM integrators are instructed to ensure that the end user has no manual instructions to remove or install the device.
FCC ID: UDV-0912142009007

This device is to be used in mobile or fixed applications only. For other antenna(s) not described in this filing the antenna gain including cable loss must not exceed 7.3 dBi in the 850 MHz Cellular band and 12.7 dBi in the PCS 1900 MHz band, for the purpose of satisfying the requirements of 2.1043 and 2.1091. The antenna used for this transmitter must be installed to provide a separation distance of at least 20 cm from all persons, and must not be co-located or operating in conjunction with other antennas or transmitters within a host device, except in accordance with FCC multi-transmitter product procedures. Compliance of this device in all final product configurations is the responsibility of the Grantee. OEM integrators and end-users must be provided with specific information required to satisfy RF exposure compliance for all final host devices and installations.

10.3. IC

Waspmote models:

Model 1- IC (XBee PRO series 1 OEM + SIM900 GSM/GPRS module )
IC: 8472A-WASP01 comprising
- IC: 4214A-XBEEPRO
- IC: 8460A-20100108007

Model 2- IC (XBee PRO ZB series 2 + SIM900 GSM/GPRS module )
IC: 8472A-WASP02 comprising
- IC: 1846A-XBEEPRO2
- IC: 8460A-20100108007

Model 3- IC (XBee 900MHz + SIM900 GSM/GPRS module )
IC: 8472A-WASP03 comprising
- IC: 1846A-XBEE09P
- IC: 8460A-20100108007

The term "IC:" before the equipment certification number only signifies that the Industry Canada technical specifications were met.

Installation and operation of any Waspmote model must assure a separation distance of 20 cm from all persons, to comply with RF exposure restrictions.

10.4. Use of equipment characteristics

- Equipment to be located in an area of restricted access, where only expert appointed personnel can access and handle it.
- The integration and configuration of extra modules, antennas and other accessories must also be carried out by expert personnel.

10.5. Limitations of use

The ZigBee/IEEE 802.15.4 module has a maximum transmission power of 20dBm.

It is regulated according to EN 301 489-1 v 1.4.1 (202-04) and EN 301 489-17 V1.2.1 (2002-08). The configuration software must be used to limit to a maximum power of 12’11dBm (PL=0).

The 868MHz XBee module has a maximum transmission power of 27dBm. This module is regulated only for use in Europe.

The 900MHz XBee module has a maximum power of 20dBm. This module is regulated only for use in the United States.

The GSM/GPRS module has a power of 2W (Class 4) for the 850MHz/900MHz band and 1W (Class 1) for the 1800MHz and 1900MHz frequency band.
The 3G/GPRS module has a power of 0,25W for the UMTS 900MHz/1900MHz/2100MHz band, 2W for the GSM 850MHz/900MHz band and 1W DCS1800MHz/PCS1900MHz frequency band.

Important: In Spain the use of the 850MHz band is not permitted. For more information contact the official organisation responsible for the regulation of power and frequencies in your country.

The cable (pigtail) used to connect the radio module with the antenna connector shows a loss of approximately 0.25dBi for GSM/GPRS.

The broadcast power at which the WiFi, XBee 2.4GHz, XBee 868MHz, XBee 900MHz operate can be limited through the configuration software. It is the responsibility of the installer to choose the correct power in each case, considering the following limitations:

The broadcast power of any of the modules added to that of the antenna used minus the loss shown by the pigtail and the cable that joins the connector with the antenna (in the event of using an extra connection cable) must not exceed 20dBm (100mW) in the 2.4GHz frequency band and 27dBm for the 868MHz band, according to the ETSI/EU regulation.

It is the responsibility of the installer to configure the different parameters of the equipment correctly, whether hardware or software, to comply with the pertinent regulation of each country in which it is going to be used.

Specific limitations for the 2.4GHz band.

- In Belgium, outdoor use is only on channels 11(2462MHz), 12(2467MHz) and 13(2472MHz) only. It can be used without a licence if it is for private use and at a distance less than 300m. Over longer distances or for public use, an I'IBPT licence is required.
- In France the use of channels 10(2457MHz), 11(2462MHz), 12(2467MHz) and 13(2472MHz) is restricted. A licence is required for any use both indoors and outdoors. Contact ARCEP (http://www.arcep.fr) for further information.
- In Germany a licence is required for outdoor use.
- In Italy a licence is required for indoor use. Outdoor use is not permitted.
- In Holland a licence is required to outdoor use.
- In Norway, use near Ny-Alesund in Svalbard is prohibited. For further information enter Norway Posts and Telecommunications (http://www.npt.no).

Specific limitations for the 868MHz band.

- In Italy the maximum broadcast power is 14dBm.
- In the Slovakian Republic the maximum broadcast power is 10dBm.

IMPORTANT

It is the responsibility of the installer to find out about restrictions of use for frequency bands in each country and act in accordance with the given regulations. Libelium Comunicaciones Distribuidas S.L does not list the entire set of standards that must be met for each country. For further information go to:

CEPT ERC 70-03E - Technical Requirements, European restrictions and general requirements: http://www.ero.dk

R&TTE Directive - Equipment requirements, placement on market: http://www.ero.dk
11. Maintenance

- In this section, the term “Waspmote” encompasses both the Waspmote device itself as well as its modules and sensor boards.
- Take care when handling Waspmote, do not let it fall, knock it or move it suddenly.
- Avoid having the devices in high temperature areas as it could damage the electronic components.
- The antennas should be connected carefully. Do not force them when fitting them as the connectors could be damaged.
- Do not use any type of paint on the device, it could harm the operation of the connections and closing mechanisms.
12. Disposal and recycling

- In this section, the term “Waspmote” encompasses both the Waspmote device itself as well as its modules and sensor boards.
- When Waspmote reaches the end of its useful life, it must be taken to an electronic equipment recycling point.
- The equipment must be disposed of in a selective waste collection system, and not that for urban solid residue. Please manage its disposal properly.
- Your distributor will inform you about the most appropriate and environmentally friendly disposal process for the used product and its packaging.