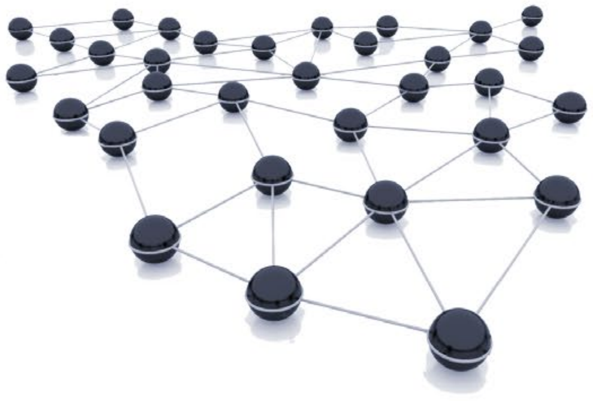
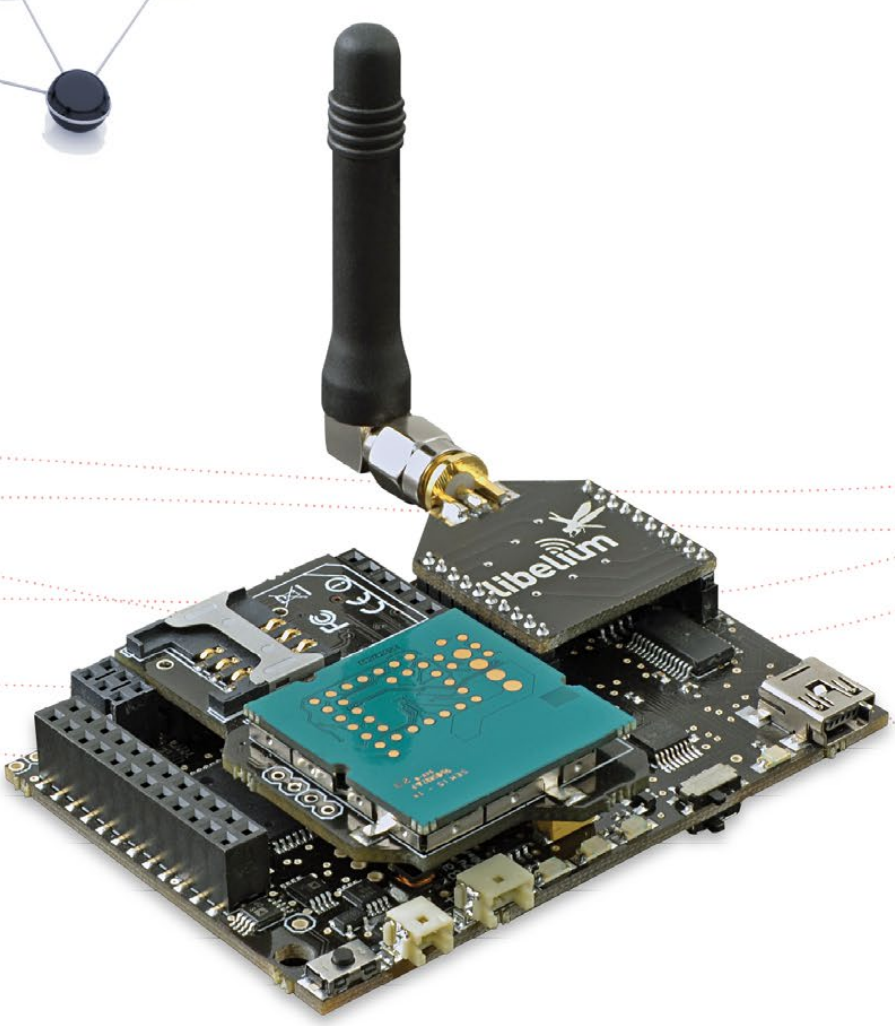
Bluetooth Low Energy

Networking Guide





Document Version: v4.1 - 06/2014

INDEX

1. Introduction

This guide will describe all features of the Waspmote Bluetooth Low Energy module, compatible with the new standard Bluetooth 4.0.

The Bluetooth 4.0 standard, also known as Bluetooth Low Energy (BLE), is a new short range radio technology, optimized for ultra low power applications. It is different from Bluetooth classic (BR/EDR), but with same benefits like robustness, interoperability, royalty free or connectivity with smart phones and PCs. Next table shows the Bluetooth terminology.

Terminology	Name	Standard	Year
BR	Basic Rate (1 Mbit/s)	1.1	2002
EDR	Enhanced Data Rate (2 and 3 Mbit/s)	2.0	2004
HS	High Speed (Alternate MAC/PHY)	3.0	2009
LE	Low Energy (1 Mbit/s ultra low power)	4.0	2010

Figure : Different Bluetooth standards

However, the compatibility with the classic Bluetooth devices depends on device trademark. See the next table which shows compatibility between Bluetooth trademarks.

Trademark	Standard version	Logo	Compatible with
Bluetooth Smart Ready (Dual mode)	v.4.0	Bluetooth® SMART READY	Bluetooth Smart Ready Bluetooth Smart Bluetooth (classic)
Bluetooth Smart (Single mode)	v4.0	Bluetooth® SMART	Bluetooth Smart Ready
Bluetooth (classic)	v2.1 + EDR, v3.0	Bluetooth®	Bluetooth Smart Ready Bluetooth (classic)

Figure : Compatibility between Bluetooth standards

Bluetooth smart ready devices are also known as *dual-mode* devices, while *Bluetooth* smart devices are also known as *single-mode* devices.

BLE modules uses the 2.4GHz band (2402MHz – 2480MHz). It has 37 data channels and 3 advertisement channels, with a 2MHz spacing and GFSK modulation.

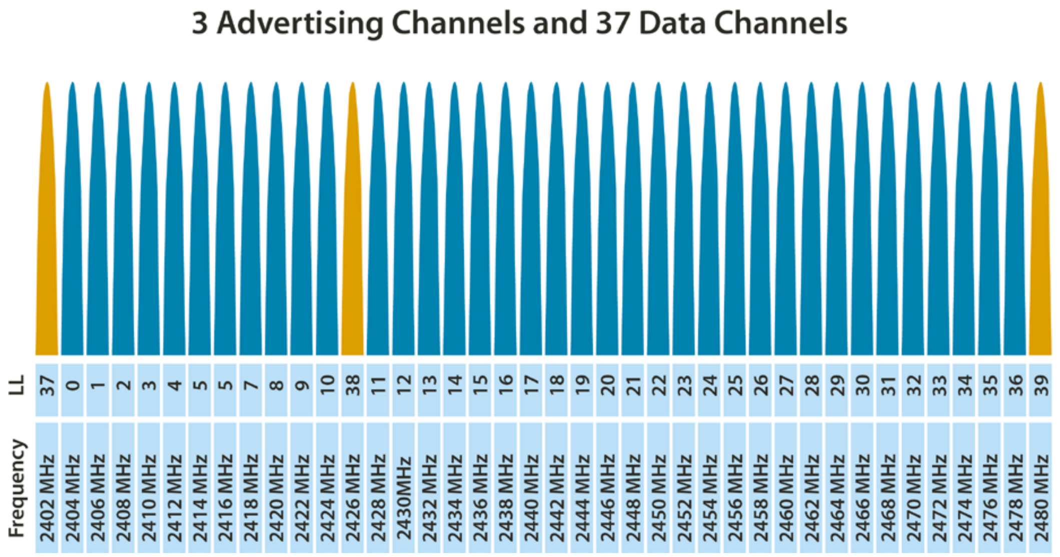

3 Advertising Channels and 37 Data Channels
LL
37 0 1 2 3 4 5 5 7 8 9 10 38 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 39
Frequency
2402 MHz 2404 MHz 2406 MHz 2408 MHz 2410 MHz 2412 MHz 2414 MHz 2416 MHz 2418 MHz 2420 MHz 2422 MHz 2424 MHz 2426 MHz 2428 MHz 2430MHz 2432 MHz 2434 MHz 2436 MHz 2438 MHz 2440 MHz 2442 MHz 2444 MHz 2446 MHz 2448 MHz 2450 MHz 2452 MHz 2454 MHz 2456 MHz 2458 MHz 2460 MHz 2462 MHz 2464 MHz 2466 MHz 2468 MHz 2470 MHz 2472 MHz 2474 MHz 2476 MHz 2478 MHz 2480 MHz

Figure : Channel distribution on the BLE standard

More information can be found at the Bluetooth SIG:

https://www.bluetooth.org/en-us/bluetooth-brand/how-to-use-smart-marks

Moreover, a dedicated API has been also created to manage the module, allowing configure main features, perform scans or connect with other devices. Next sections will describe it in deep, but if the user requires more information about the internal chip set, a lot of literature is provided at the manufacturer website, which can be useful to understand basic concepts of the BLE technology.

2. Hardware

2.1. Specifications

The BLE module is managed by UART and it can be connected on SOCKET0 and SOCKET1 of Waspmote. The main features of the module are listed below:

- Protocol: Bluetooth v.4.0 / Bluetooth Smart
- Chipset: BLE112
- RX Sensitivity: -103dBm
- TX Power: [-23dBm, +3dBm]
- Antenna: 2dBi/5dBi antenna options
- Security: AES 128
- Range: 100 meters (at maximum TX power)
- Consumption: sleep (0.4uA) / RX (8mA) / TX (36mA)

Actions:

- Send broadcast advertisements (iBeacons)
- Connect to other BLE devices as Master / Slave
- Connect with Smartphones and Tablets
- Set automatic cycles sleep / transmission
- Calculate distance using RSSI values
- Perfect for indoor location networks (RTLS)
- Scan devices with maximum inquiry time.
- Scan devices with maximum number of nodes.
- Scan devices looking for a certain user by MAC address.

Figure : Waspmote Bluetooth Low Energy module

2.2. Electrical characteristic and power consumption

The Libelium BLE module is powered from 3.3V. Next table shows the average power consumption in different states of the module.

State	Power consumption
OFF	0mA
Sleep	0.4uA*
ON	8mA
Transmitting (advertising / data connection)	36mA

**Consumption in best low power mode. See manufacturer documentation for more information.*

Figure : Power consumption at different states.

The Bluetooth Low Energy standard has been designed for low power consumption. Despite of the peak current during transmission can reach relatively high values, the user should take into account that these values are present only for small periods of time. Next graph shows an average curve of the current consumption (in mA) versus time (in ms) while one advertisement is sent by the BLE module.

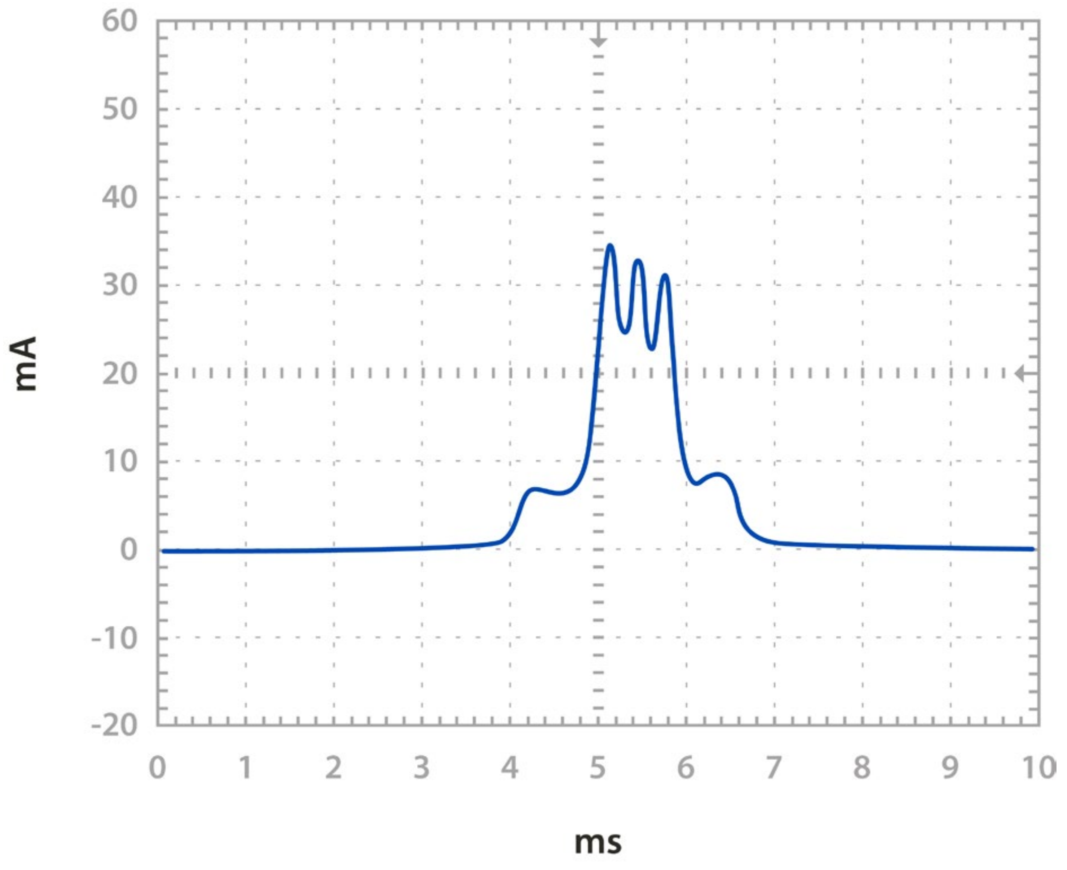


Figure : Average current consumption during one advertisement

2.3. Auxiliary connector

Note: *The usage of this connector is only recommended for advanced users.*

Besides the UART interface used with Waspmote, the manufacturer provides other optional functions for the module. The BLE module auxiliary connector includes some of these functions to allow the user developing his own application using I/O pins or USB interface.

It is important to say that these functions are not directly used by Waspmote, so the user would have to develop his own application. The usage of the auxiliary connector involves sending specific commands defined in the manufacturer documentation. For this reason, its usage is only recommended for experienced users.

The pin out of auxiliary connector is described below.

Pin number	1	2	3	4	5	6	7	8	9	10
Internal connection	USB+	USB-	P0.7	P0.3	P0.2	P0.5	P0.4	Not used	Not used	GND

Figure : Auxiliary connector

Important: *The connection of any extra hardware through the auxiliary connector can damage the module or even Waspmote if it is not done correctly. Any damage caused by external hardware will void the warranty.*

3. General Considerations

This section will describe the BLE module API. The functions which manage BLE module belong to the class **WaspBLE**, and the object created to use them is defined as **BLE**.

The module talks to Waspmote using a binary protocol through a serial interface. When a command is sent, the module answers back acknowledging the command, and also commands can produce events. The implemented functions manage the commands, answer and events to make easier handling the module. The diagram below shows the described process.

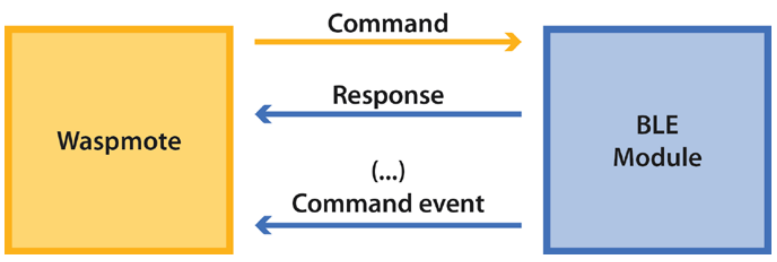


Figure : Serial communication between Waspmote and BLE module

Besides that, the Waspmote API can manage the module to switch between the four device roles defined by the Bluetooth Standard.

- **Advertiser:** Broadcast advertisements
- **Scanner:** Listen for advertisements. Can connect to an advertiser.
- **Master (central):** Communicates with the device in slave role, defining timings and transmissions.
- **Slave (peripheral):** A device connected to a master. Can have only one master at the same time.

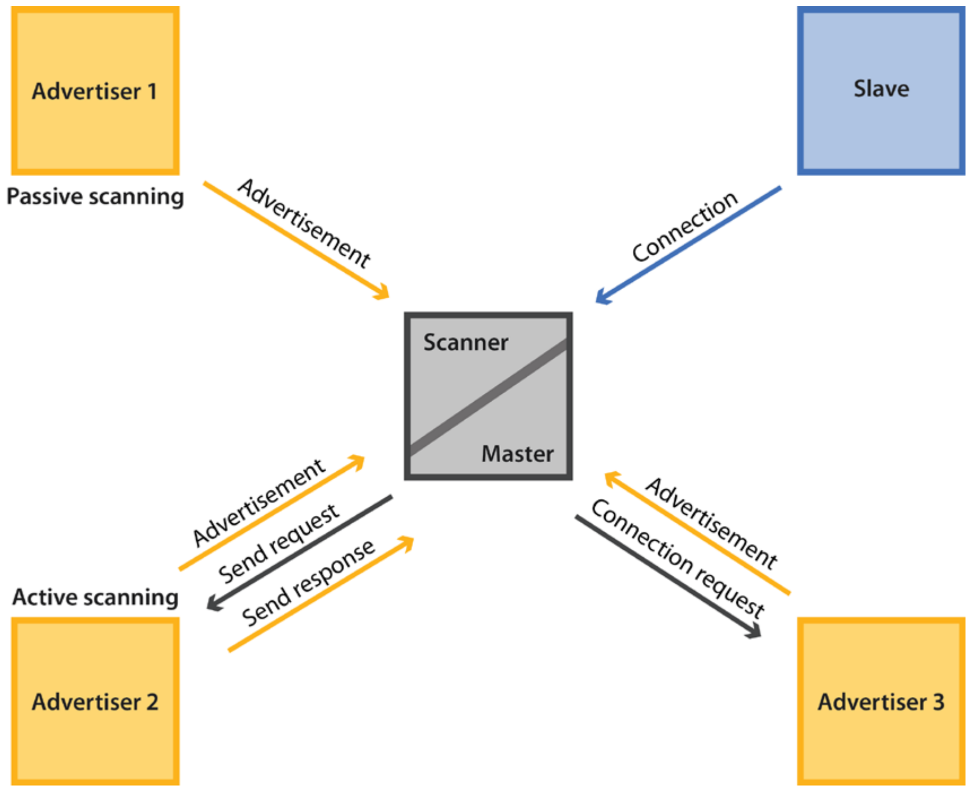


Figure : Device roles and relationship between them

Full information is provided in next sections, including some examples of use and some Bluetooth standard concepts necessary to understand the main features of the module. However, if the user requires more information about the Bluetooth standard, please visit the Bluetooth SIG website: **http://www.bluetooth.org**.

3.1. Waspmote Libraries

3.1.1. Waspmote BLE Files

```
WaspBLE.h; WaspBLE.cpp
```

It is mandatory to include the BLE library when using this module. So the following line must be added at the beginning of the code:

```
#include <WaspBLE.h>
```

3.1.2. Constructor

To start using the Waspmote BLE library, an object from class WaspBLE must be created. This object, called BLE, is already created by default inside Waspmote BLE library. It will be used through this guide to show how Waspmote BLE library works.

When creating this constructor, all the variables are defined with an initial value by default. Some of these variables are:

- `_baudrateBT:` specifies the baudrate used to communicate with the module (115200 bps by default)
- `errorCode:` saves error codes returned by the module.

3.2. API functions

Through this guide there are many examples of using functions. In these examples, API functions are called to execute the commands, storing in their related variables the parameter value in each case.

Example of use

```
{
  BLE.getOwnMac(); // Get BLE module's MAC address
  BLE.getScanningParameters(); // Get current scanning parameters
}
```

Related Variables

`BLE.my_bd_addr` → Stores the BLE module MAC address

`BLE.GAP_discover_mode` → Stores General access profile discover mode

`BLE.scan_interval` → Stores at what intervals scanner is started

`BLE.scan_window` → Stores how long to scan at each scan interval

`BLE.scan_duplicate_filtering` → Stores if active or passive scan is taking process.

`BLE.TXPower` → Stores TX power

When returning from `BLE.getOwnMac`, the related variable `BLE.my_bd_addr` will be filled with the appropriate value. Before calling the function, the related variable is created but it is empty or with a default value.

All the functions also return a flag to know if the function called was successful or not. The most common case is to return the `BLE.errorCode` variable which contains the answer of the module to the command sent. Detailed information about the meaning of each error code can be found in the manufacturer documentation.

3.3. API extension

All the relevant and useful functions have been included in the Waspmote BLE API, although any command can be sent directly to the BLE module with the `sendCommand` function. The command can be sent as a string or as an array of uint8_t, adding the total length. "Packet mode" is used in Waspmote API, so the first byte must be the command length.

Example of use

```
{
  // sending "hello" command
  BLE.sendCommand("0400000001");

  // sending the command as an array of uint8_t
  uint8_t command = {0x04, 0x00, 0x00, 0x00, 0x01};
  BLE.sendCommand(command,5);
}
```

Sending commands example:

http://www.libelium.com/development/waspmote/examples/ble-20-sending-custom-commands

3.4. Waspmote reboot

When Waspmote is rebooted the application code will start again, creating all the variables and objects from the beginning.

3.5. Constants predefined

There are some constants defined in a file called `WaspBLE.h`. These constants define some parameters like the baudrate used for serial communication. The most important constants are explained next:

Constant	Default value	Description
`BLE_DEBUG`	0	Debug mode label. Possible values are: 0 = no debug messages will be printed 1 = some debug messages will be printed 2 = all messages of internal communication will be printed
`ENABLE_EEPROM_SAVING`	1	Comment this line to avoid using EEPROM with this library.
`BT_BLUEGIGA_RATE`	115200	Baudrate used for serial communication between Waspmote and BLE module.
`MAX_PACKET_SIZE`	65	Maximum packet size allowed by the protocol.
`DEFAULT_SCAN_TIME`	30	Default scanning time when it is not specified.
`TX_POWER_MAX`	15	TX power range from 0 to 15, equivalent to power from -23 to +3 dBm.

There are other less relevant constants for secondary functions or internal library usage. They can be found at the header file `WaspBLE.h`.

4. Initialization

Before start using the module, it needs to be initialized. During this process, the UART to communicate with the module has to be opened, module is powered and some variables are initialized.

4.1. Setting ON

With the function `ON` the module is powered and the UART is opened. It returns '1' on success, '0' if error.

Besides, default configuration parameters are sent to the Bluetooth module.

Example of use:

```
{
 BLE.ON(); // starts power and configuration parameters
}
```

4.2. Setting OFF

Bluetooth function OFF closes the UART and switches off the module.

Example of use:

```
{
 BLE.OFF(); // switches the module off
}
```

4.3. Sleep mode

The module has a sleep mode to reduce power consumption when UART interface is not needed. Basically, this function disables UART interface, allowing the module to enter in the Bluetooth low power modes. Waspmote and its sleep modes are independent from the BLE sleep mode.

There are three low power modes, which are automatically managed by the module, so **the user has no control on them**. The module has been optimized by the manufacturer to select the best power mode for each condition.

Power Mode	Consumption
Power mode 1	235 uA
Power mode 2	0.9 uA
Power mode 3	0.4 uA

Figure : Low power modes of the BLE module

The function to enter in sleep mode is `Sleep`. When this function is called, the module will enter in the best power mode available, and it will continue advertising, or being visible if the user has previously enabled these features.

Example of use:

```
{
 BLE.sleep(); // puts the BLE module into sleep mode
}
```

The function to leave sleep mode and enable again UART interface is wakeUp.

Example of use:

```
{
 BLE.wakeUp(); // wakes up BLE module
}
```

In the Waspmote Development section you can find a complete example about using this function.
Go to: **http://www.libelium.com/development/waspmote/examples/ble-18-sleep-mode**

4.4. Reset

If a software reset is needed for the BLE module, then the reset function would be used.

Example of use:

```
{
  BLE.reset();
}
```

In the Waspmote Development section you can find a complete example about using this function.
Go to: **http://www.libelium.com/development/waspmote/examples/ble-21-software-reset**

5. Scanning BLE devices

The way of detect other Bluetooth nodes is making a scan looking for slave messages (called advertisements). Some parameters should be configured to allow scanning in the desired way. Besides that, some scanning functions have been implemented to easy scan for other devices, defining time, number of devices to discover, etc.

5.1. Configuring a scan

Next parameters can be configured before doing a scan. The parameters are defined inside the General Access Profile (GAP) of the module. It is not possible to change scanning parameters while scanning, so the user must stop scanning before making any change.

General Access Profile mode:

There are three possible states regarding **visibility** of BLE devices, so the scan can filter the devices by this state. The function `setDiscoverMode` is used to manage them. The possible choices are:

- Limited discoverable: Discover only slaves which have the limited discoverable mode enabled.
- Generic discoverable: Discover slaves which have limited discoverable mode or generic discoverable mode enabled.
- Observation: Discover all devices (default).

Example of use:

```
{
 BLE.setDiscoverMode(BLE_GAP_DISCOVER_OBSERVATION);
}
```

Scan Interval

Defines at what intervals scanner is started. This variable is defined in units of 625us and and has a range between 4 and 16384. By default is set to 75, which is equivalent to 75 * 625 us = 46.875 ms. The `SetScanningParameters` function is used for this purpose.

Example of use:

```
{
 // set scan interval to 100 ms (160 * 0.625 = 100)
 BLE.setScanningParameters(160,80,0);
}
```

Scan Window

Defines how long to scan at each interval. This variable is defined in units of 625us and and has a range between 4 and 16384. By default is set to 50, which is equivalent to 50 * 625 us = 31.250 ms. It must be equal or smaller than scan interval.

Example of use:

```
{
 // set scan window to 50 ms (80 * 0.625 = 50)
 BLE.setScanningParameters(160,80,0);
}
```

Passive / Active scanning

This parameter selects between two types of scanning: passive and active.

In passive scanning, the BLE module just listens to other node advertisements. When one of these advertisements is detected, the module reports to Waspmote the discovered device. Normally, the advertisement contains information like discoverability and connectability modes, TX power level, MAC address of the node (called advertiser) or application data.

On the other hand, in active scanning the module will request more information once an advertisement is received, and the advertiser will answer with information like friendly name and supported profiles. The advertisement packets are configurable as is described in next sections.

Another prototype of `setScanningParametrs` is defined if only scanning is being changed.

Example of use:

```
{
  // setting active scanning
  BLE.setScanningParameters(BLE_ACTIVE_SCANNING);
}
```

MAC Filtering and scan policy

The BLE module offers the possibility of sending one report for each node detected (MAC filter enabled) or, on the other hand, send multiple events for each node detected. By default, MAC filter is enabled for all scanning functions.

Besides that, the module has the possibility to accept advertisements only of specific nodes defined inside a list (called "white list"), by defining a scan policy. By default, the scan policy accepts all incoming advertisements. The API function which allow this configuration is `setFiltering`. The second parameter called advertising policy is described later, because it is related with advertisements.

Example of use:

```
{
  // enabling MAC filter and setting scan policy to allow scan all devices.
  BLE.setFiltering(BLE_GAP_SCAN_POLICY_ALL, advertisement_policy, BLE_MAC_FILTER_ENABLED);
}
```

TX power

The Transmitting power can be set by using `setTXPower`, using a value value between 0 and 15 which give the real TX power from -23 to +3 dBm, according to the hardware of the module. Next graph from the manufacturer shows this relation.

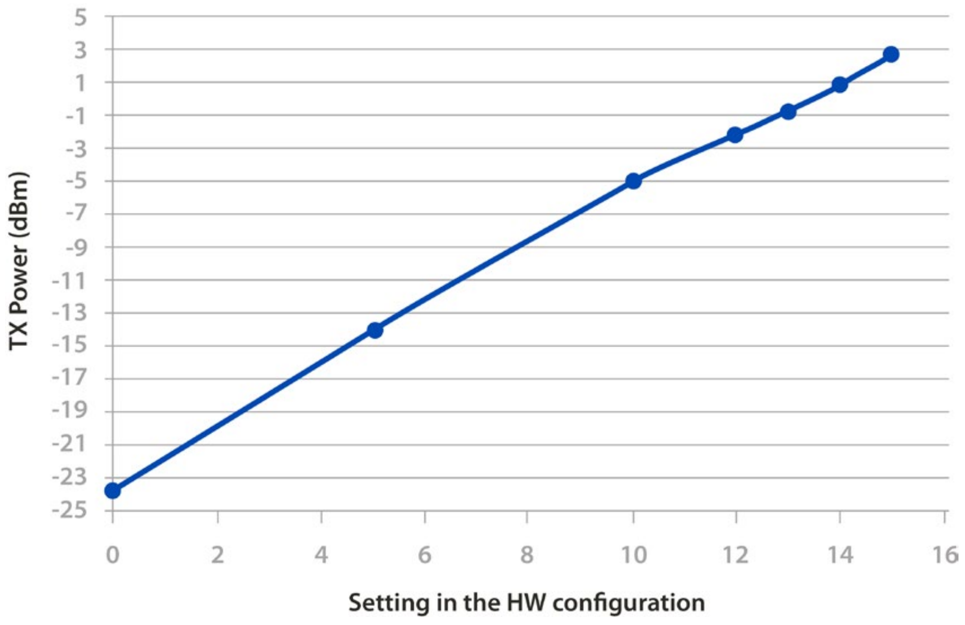


Figure : TX power values related with real TX power. Figure provided by the manufacturer.

Example of use:

```
{
  // setting maximum TX power (+3 dBm)
  BLE.setTXPower(TX_POWER_MAX);
}
```

In order to know the current scanning parameters, the function `getScanningParameters` prints by USB all parameters involved in scanning BLE modules. Some examples of scan configurations are listed below.

Example 1

Making a scan with default parameters: Observation scan, interval scan of 46.875 ms, scan window of 31.250 ms, passive scanning, MAC filter enabled and maximum TX power.

```
{
  // It is not necessary to configure anything
  BLE.scanNetwork(5);
}
```

Example 2

Configuring a generic scan of 10 seconds with TX power of -5 dBm, active scanning enabled, MAC filter enabled, with a scan interval of 100 ms and a scan window of 50 ms.

```
{
  BLE.setDiscoverMode(BLE_GAP_DISCOVER_GENERIC);
  BLE.setTXPower(10);
  BLE.setScanningParameters(160, 80, BLE_ACTIVE_SCANNING);
  BLE.scanNetwork(10);
}
```

5.2. Scan functions

There are some predefined functions to make scans, like in the BT_Pro module, allowing to specify time for scanning or a device to find. These functions automatically parses other node advertisements detected by the BLE module and save the data if necessary. See below the data structure of an advertisement detected by the BLE module, and the description of the scanning functions.

Advertisement									
Msg type	Payload	Msg Class	Msg ID	RSSI	Packet type	Sender	Address type	Bond	Data

Figure : Advertisement structure reported by the BLE module acting as a scanner

ScanNetwork

This function allows a simple scan of the network. The time of the scan duration must be specified in seconds. Besides that, another prototype has been defined to allow specifying also the TX power for the scan.

If EEPROM usage is enabled by the label `ENABLE_EEPROM_SAVING`, the scan results will be saved into it, and they can be shown using the function `printInquiry`, which prints by USB the data of the last scan. The EEPROM addresses used are defined by the labels `SCAN_EEPROM_START_ADDRESS`, `SCAN_EEPROM_LIMIT_ADDRESS` and `SCAN_END`. If the user does not want to use the EEPROM for this purpose, then a new storing way has to be defined (by global buffers, using SD card, etc). More information about EEPROM usage is provided in next sections. Besides that, the variable `numberOfDevices` stores the total number of discovered devices during the scan.

Example:

```
{
  // scan network during five seconds
  BLE.scanNetwork(5);

  // scan network specifying also the TX power
  BLE.scanNetwork(5, TX_POWER_MAX);
}
```

In the Waspmote Development section you can find a complete example about using this function.

Go to: **http://www.libelium.com/development/waspmote/examples/ble-01-normal-scan**

ScanNetworkName

This function scans the network to find other BLE modules, including friendly name of each module. It uses active scanning, because the friendly name is not sent by default on the passive advertisements. The time of the scan duration must be specified in seconds. Besides that, another prototype has been defined to allow specifying also the TX power for the scan.

Example:

```
{
  // scan network during five seconds, including friendly name
  BLE.scanNetworkName(5);

  // scan network specifying the TX power, including friendly name
  BLE.scanNetworkName(5, TX_POWER_MAX);
}
```

If the scanned devices have no friendly name available, the friendly name field will be empty.

In the Waspmote Development section you can find a complete example about using this function.

Go to: **http://www.libelium.com/development/waspmote/examples/ble-02-name-scan/**

ScanNetworkLimited

This function scans the network in the same way of `scanNetwork`, but now you can limit your scan to a specific number of BLE modules. If this number is reached, the inquiry stops. However, if maximum is not reached, the module continues inquiring until `DEFAULT_SCAN_TIME`.

The number of devices to discover must be specified. Besides that, another prototype has been defined to allow specifying also the TX power for the scan.

Example:

```
{
  // scan network till find two devices
  BLE.scanNetworkLimited(2);

  // scan network till find two devices, setting also TX power
  BLE.scanNetworkName(5, TX_POWER_MAX);
}
```

In the Waspmote Development section you can find a complete example about using this function.

Go to: **http://www.libelium.com/development/waspmote/examples/ble-03-limited-scan/**

ScanDevice

This function makes a scan to discover a specific BLE device by its MAC address. This function is used to know if a specific device is present inside detection area of the BLE module. Devices are commonly identified by its MAC address, which is a number of 12 digits, for example "00078053C1B9".

The node MAC address must be specified as a parameter, but it has been defined another prototype to specify also the maximum scanning time and the TX power. If the user does not specify these parameters, default values will be used.

Example:

```
{
 // scan network till find specified device
 BLE.scanDevice("00078053C1B9");

 // Look for a specified device, during given time and using given TX power
 BLE.scanDevice("00078053C1B9", 5, TX_POWER_MAX);
}
```

In the Waspmote Development section you can find a complete example about using this function.

Go to: **http://www.libelium.com/development/waspmote/examples/ble-04-scan-device/**

PrintInquiry

By default, discovered devices are not shown through USB because they are only saved on EEPROM memory (if enabled). If the user wants to view discovered devices just use `printInquiry` function after the scan function and discovered devices in the last inquiry will be shown. It also returns the number of devices printed.

Example:

```
{
 // scan network for 5 seconds
 BLE.scanNetwork(5);
 USB.printf("discovered devices: %d\r\n", BLE.numberOfDevices);

 // Now print last inquiry saved on EEPROM
 USB.println("Printing Last inquiry saved on EEPROM:");
 uint8_t dummy = BLE.printInquiry();
 USB.printf("devices printed:%d\r\n\r\n",dummy);
}
```

Finally, if the user want to cancel an inquiry on a certain moment, `scanNetworkCancel` function will stop any scan in course.

5.3. EEPROM management

The EEPROM memory of Waspmote can be used to store scan results. By default, this feature is enabled, so the user should take into account if other parts of the application are using it. A certain zone of the EEPROM memory is reserved for BLE module usage, defined with the next labels

- `SCAN_EEPROM_START_ADDRESS`
- `SCAN_EEPROM_LIMIT_ADDRESS`

By default, 800 bytes are reserved (enough for 20 devices), starting on position 1024 and ending at 1823, but the user is free to enlarge or move the reserved space to other EEPROM zones, even change the way of storing scanned devices to select only certain information. Moreover, the user can easily disable this feature by setting the label `ENABLE_EEPROM_SAVING` to 0.

More in detail, a scan network is saved using the structure `discoveredDevice` shown below:

```
struct discoveredDevice

     {
      uint8_t mac[6];             // MAC Address of the BLE device
      int8_t rssi;                // Received signal strength of the BLE device
      uint8_t friendlyName[31];  // Friendly name of the BLE device
      uint8_t advData[33];        // Advertisement data of the BLE device
     };
```

Where each field contains information about the last scanned BLE device, like the MAC address, the RSSI (Received Signal Strength Indicator, in dBm and with a range from -103 to -38), the friendly name and the advertisement data. An end sequence is used to know the scan data length. The friendly name is filled with either short friendly name or large friendly name, as defined in the Bluetooth 4.0 standard. If no friendly name is found for the scanned device, the field will be empty. Each scan can be easily printed by the function `printInquiry`, which prints by USB the last scan saved.

Finally, the advertisement data stored in `advData` is not saved in memory. The user will have to develop its own application and use this data in the preferred way.

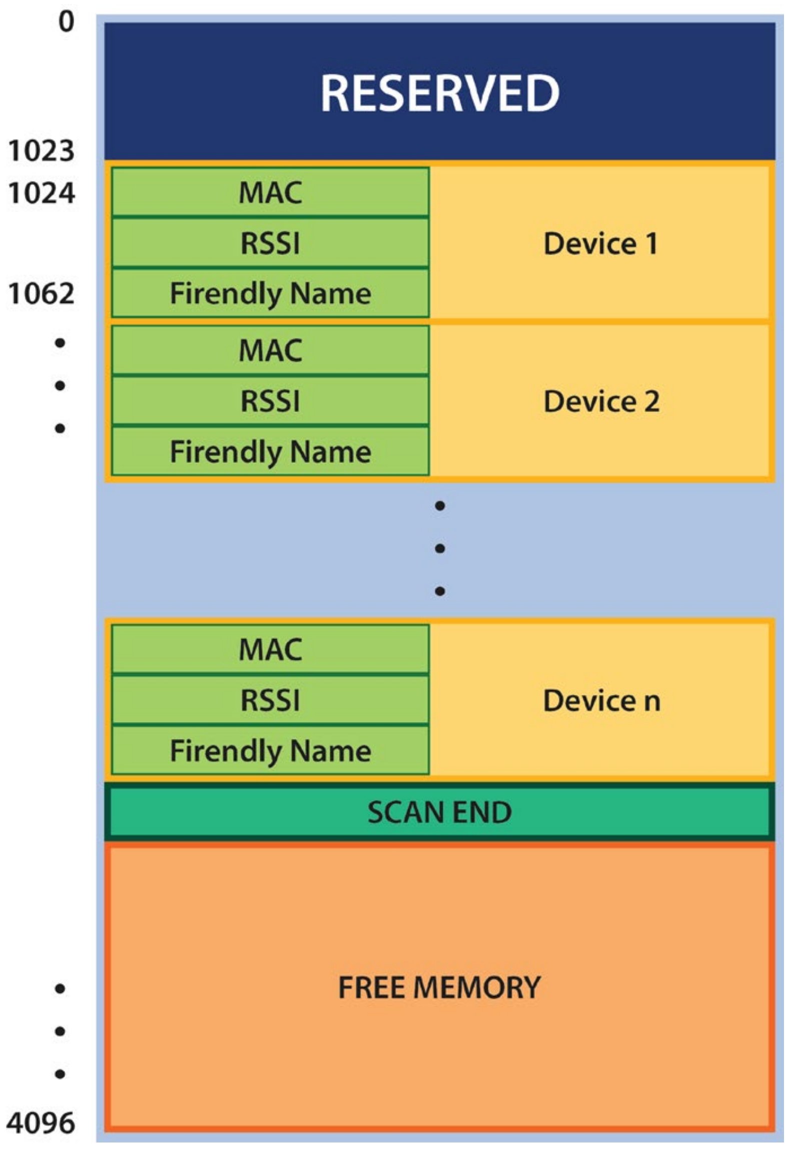


Figure : EEPROM usage by the BLE API

5.4. Relationship between RSSI and distance

As described in previous section, each scanned device has an associated RSSI value in dBm. This value is directly related with the distance of the scanned node from the scanner.

Besides that, the user can modify the TX power used for the scan, and using both properties, the Waspmote BLE module can be used to know if a BLE device is too close to the scanner, in a middle range or far from the scanner, performing different actions on them. Next graph shows this relationship, where a BLE node is approaching the BLE scanner. Average RSSI values of a BLE node are faced to the distance from the BLE scanner.

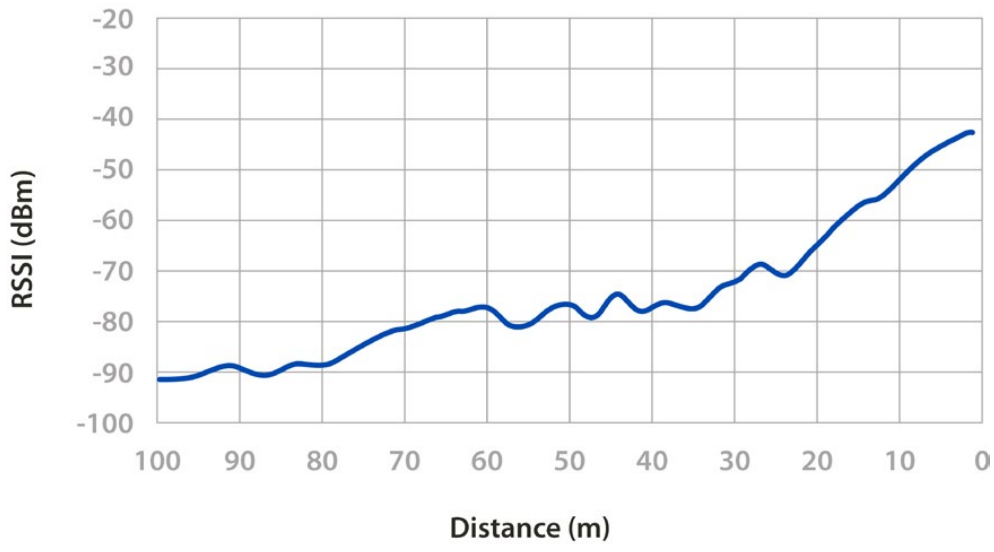


Figure : Relationship between RSSI and distance.

5.5. Indoor location

The relationship between the RSSI and the distance between two BLE nodes can be used for indoor location applications. Placing various BLE nodes inside a room, it is easy to obtain some RSSI values of the same BLE device, allowing the estimation of the BLE exact location inside the room.

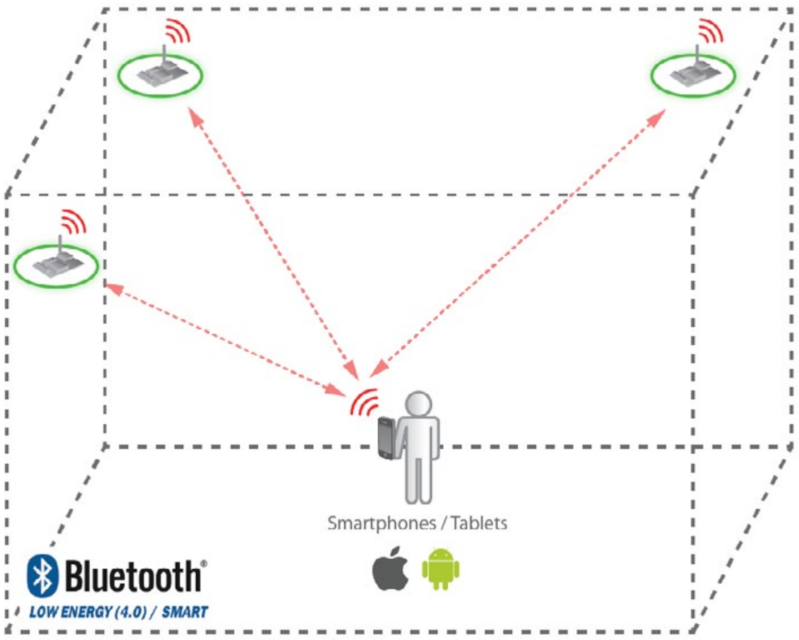


Figure : Indoor location application example

6. Advertising

Advertising is a process used for BLE modules to send data to other BLE devices. This data is packed in advertisements, which are modifiable by the user, so they represent an easy way of sending data without the need of connecting first.

The advertisements contains information about address of the advertiser, discoverability and connectability modes, TX power level, application data, etc. They can be built according to the Bluetooth standard or customized by the user to send its own data, as It is described later.

6.1. Configuring advertisements

There are some parameters which allow the advertisement configuration. For example, the user will be able to configure which nodes can see the Waspmote BLE module advertisements. Below all possibilities are described.

Visibility - GAP discoverable modes

The way of controlling who is able to see the advertisements of a BLE node is modifying the GAP discoverable mode with the function `setDiscoverableMode`. Possibilities are shown below

- *Non discoverable:* Device is not advertising
- *Limited discoverable:* Device will be visible using limited discoverable scanning mode
- *General discoverable:* Device will be visible using general discoverable scanning mode
- *Broadcast:* Same as non discoverable
- *User Data:* Send advertisement data defined by user
- *Enhanced broadcasting:* scanning devices are reported back to the application.

Example:

```
{
  // start advertising in general discoverable mode
  BLE.setDiscoverableMode(BLE_GAP_GENERAL_DISCOVERABLE);
}
```

Conectability – GAP connectable modes

The user can also specify who can connect to the node, setting a flag inside the advertisement with the choices listed below. It can be done also with function `setConnectableMode`, possibilities are:

- *Non connectable:* Device is not connectable by others
- *Direct connectable:* Device can be connected by only one node
- *Indirect connectable:* Device can be conne cted by any node

Example:

```
{
   // Allow connections from everyone.
   BLE.setConnectableMode(BLE_GAP_UNDIRECTED_CONNECTABLE);
}
```

Advertising policy

It is also possible to define an advertising policy to configure if the module will answer to the scan requests and will allow connections for other devices. It is done with function `setFiltering`. Four choices are possible:

- *All:* Responds to scan requests from any master, allows connection from any master.
- *WhiteList scan:* Responds to scan requests from whitelist only, allows connection from any master.
- *WhiteList connect:* Responds to scan requests from any master, allows connection from whitelist only.
- *WhiteList all:* Respond to scan requests from whitelist only, allows connection from whitelist only.

The `setFiltering` function also allows defining scan policy and MAC filter, as discussed previously.

Example:

```
{
        /* Example where all present BLE modules are scanned, but only whiteList  members
are allowed to conect. Also MAC filter is enabled to produce one     event per device.
  */
        BLE.setFiltering(BLE_GAP_SCAN_POLICY_ALL,      BLE_GAP_ADV_POLICY_WHITELIST_CON-
NECT,
  BLE_MAC_FILTER_ENABLED);
}
```

TX power

Changes in the TX power will affect to the range of node advertisements. As described previously, it can be changed using the `setTXPower` function.

Advertisement intervals and allowed channels

The time between advertisements can be also configured according to two parameters:

- advertisement interval minimum.
- advertisement interval maximum.

They are defined in units of 625 uS, with a configurable range between 20 ms and 2.5 seconds. Default values are set to 320 ms.

On the other hand, the BLE standard uses three channels for advertising, which are 37, 38 and 39. The user can select which channel is used with a bitmask. For example, enable three channels with 7 (binary 0111), enable channels 37 and 39 with 5 (binary 0101) and so on.

If the module is advertising, any changes on these parameters will only take effect stopping advertising and start again.

Example of use:

```
{
  // setting advertisements each 100 ms on the three channels at same time
  BLE.setADVParameters(160,160,7);
}
```

6.2. Advertisement data structure

The maximum size of advertisement data is **31 bytes**. This is the maximum size allowed by the Bluetooth 4.0 specification. The user can build the advertisement with a custom payload, but if compatibility with Bluetooth 4.0 specification is required, the data must be formatted according to the core specification. Visit the official Bluetooth SIG website for detailed information: **http://www.bluetooth.org**

Two types of advertisements can be configured on the BLE module: normal advertisements and scan responses. The normal advertisements are sent when the discoverable mode is different from non discoverable, in response to passive and active scans, while the scan response advertisements are sent only when an active scan is performed.

The advertisement data can be set using the function `setAdvData`, specifying if normal advertisement or scan response is being modified, the data to place in the payload and its length. If the user introduces a data larger than allowed, advertisement data will not be set.

Example of use:

```
{
  // create an array to store adv data and fill it with user data.
  uint8_t data[31];

  // set normal advertisement data
BLE.setAdvData(BLE_GAP_ADVERTISEMENT, data, 31);

// set scan response data
BLE.setAdvData(BLE_GAP_SCAN_RESPONSE, data, 31);
}
```

In the Waspmote Development section you can find a complete example about using this function.

Go to: **http://www.libelium.com/development/waspmote/examples/ble-17-configuring-advertisements**

7. Connecting Waspmote to other BLE devices

Besides advertising data, there is a more robust and reliable way of transmitting data. The data sent using connections processes can be acknowledged and encrypted.

When a BLE device acting as scanner detects advertisements of a BLE node in advertiser role, the scanner can try to connect the advertiser sending a connection request with the function `connectDirect`. If the connection is successful, the scanner will become the master (or central) of the connection, and the advertiser will be the slave (or peripheral).

According to the Bluetooth standard, a slave can be connected only to a single master, while a master can be connected to multiple slaves.

7.1. Configuring a connection

The user can easily connect using the `connecDirect` function and providing the MAC address of the BLE node to be connected. If this option is used, default connection parameters will be used.

Example of use:

```
{
 BLE.connectDirect(“0007806AF308”) ;
}
```

However, the connection parameters can be also configured, as explained below. It is important to say that connection parameters can be updated during the connection.

Connection intervals

The minimum and maximum connection intervals can be specified also with the `connectDirect` function, but using other prototype defined for that purpose. The lowest connection interval is 7.5 ms, and the maximum is 4 s. The maximum connection interval must be equal or greater than the minimum.

Supervision timeout

This is the timeout to wait till drop the connection, in units of 10 ms. Range between 10ms and 32 seconds.

Connection latency

This parameter configures how many connection intervals can be skipped by the slave. It is useful in scenarios where data is not needed every connection interval, to save power on the slave side. Latency range goes from 0 to 500.

For example, if the user sets a connection interval of 10 ms and a latency of 9, then the slave will communicate with the master each 100 ms, but it can communicate each 10 ms if needed.

On the other hand, slave latency * connection interval can not be bigger than supervision timeout.

Example of use:

```
{
 BLE.connectDirect(BLEAddress,
                   conn_interval_min,
                   conn_interval_max,
                   timeout,
                   latency)
}
```

Connection handle

When the master opens a connection, each connection is identified by a handler, saved on variable `connection_handle`.

Connection RSSI

The RSSI of the connection can be known by the function `getRSSI`. The range is the same then the defined for the scanned devices.

Disconnection

When the user wants to terminate a connection manually, it can be done using the `disconnect` function and the corresponding connection handle.

Example of use:

```
{
  BLE.disconnect(BLE.connection_handle);
}
```

Besides that, the user can now in every moment the status of the BLE module with the `getStatus()` function.

Example of use:

```
{
  BLE.getStatus();
}
```

The connection parameters are illustrated in the diagram below.

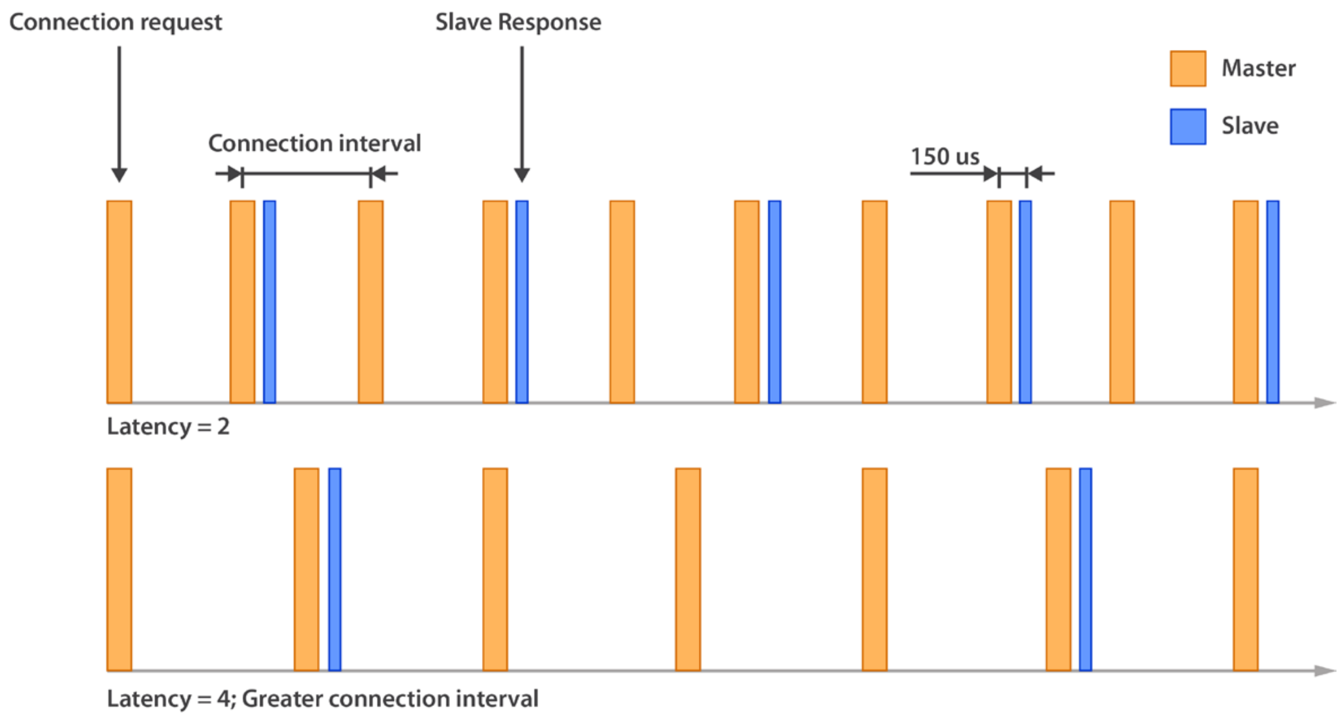


Figure : Connection parameters

In the Waspmote Development section you can find a complete example about using this function.

Go to: **http://www.libelium.com/development/waspmote/examples/ble-06-configuring-a-connection**

7.2. Sending data

There are two terms used when talking about the two members of a connection.

- **Client:** Device which accesses a remote BLE. Usually the master is also the client. Clients are usually smart phones, tablets or PCs.
- **Server:** Device with a local database that can be accessed by the remote client. The slave can be also the server. Servers are usually sensor nodes, like thermometers or hear rate sensors.

When a connection is established, there are four operations to move data between clients and servers. All of them are based of moving data of the characteristics defined inside the slave profile.

- **Read:** This operation is requested by the client, on a specific server attribute. The server responds with the requested data.
- **Write:** This operation is requested by the client, on a specific server attribute. The server stores the value sent by the client and it can optionally confirm the operation if notify / indicate properties are enabled.
- **Notify:** This operation is started by the server. When a client is subscribed to notifications on a certain characteristic, it will be informed immediately when a new value is written by the server on this attribute. The figure below shows the process.
- **Indicate:** This operation is equal to notify, but the entire process is acknowledged with a client message to the server.

In resume, the client can use read, write, notify and indicate operations to access data on a remote server, while the server stores the data locally and provides data access methods to a remote client. Next figure illustrates the process during a write operation when notifications and indications are enabled. Each step is also described.

Step 1

Once the connection is established, the master subscribe to a certain slave characteristic with a Notify / Indicate subscription.

Step 2

Waspmote writes into the BLE module (Slave) a new value on a local characteristic, using the function `writeLocalAttribute`, and this change is notified to the master automatically.

Step 3

The difference between notify and indicate operations resides on this step. An acknowledge event is sent to the slave to indicate that the change on the value has been received. In this way, Waspmote can know if the master has received the information.

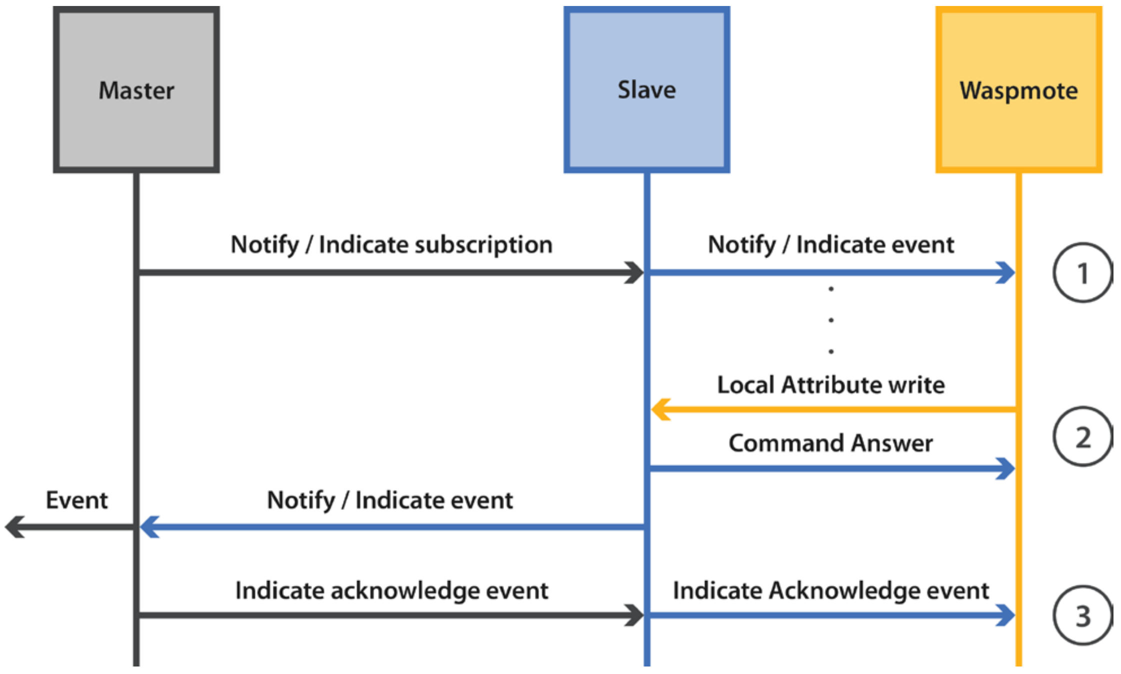


Figure : Notification and indication processes

Simple Read / write operations are thought when data is not needed to be acknowledged. However, It has to be clarified that read/write operations can be performed in two different ways: under own attributes and under slave attributes (through a connection).

In the first way, the node can read/write its own attributes defined in its profile, using the functions `readLocalAttribute` and `writeLocalAttribute`. In case of indicated writes (acknowledged) another prototype of `writeLocalAttribute` has been defined. Maximum data size for local read /write operations is 32 bytes.

Example of use:

```
{
 // write a local attribute
 BLE.writeLocalAttribute(handle, data);
 // write a local attribute with indication.
 BLE.writeLocalAttribute(handle, BLE_ENABLE_INDICATE, data);
}
```

The other way consist of establish a connection with a slave and read/write its attributes remotely, therefore the local attributes will not be changed. It can be done using the functions `attributeRead` and `attributeWrite`. Maximum data size for remote read operations is 22 bytes, while write, notify and indicate are 20 bytes. The user can use these functions to subscribe to Notifiable / Indicable characteristics.

Example of use

```
{
 // a connection with the device is required.
 if(BLE.connectDirect(“0007806AF308”) == 1)
 {
        // read an attribute
        BLE.attributeRead();
        // save the read attribute
        (..)

        // write an attribute
        BLE.attributeWritte(connection, handle, data);
 }
}
```

When a read operation (local or remote) is done, the attribute value read is stored in the global variable `attributeValue`. The user should save this value before reading or writing other attribute, otherwise it will be lost.

In the Waspmote Development section you can find some complete examples about using these functions.

Go to:
http://www.libelium.com/development/waspmote/examples/?cat=communication-examples&subcat=bluetooth-low-energy

7.3. Encryption and Bonding

In older Bluetooth standards, the user needed to pair with a remote device before connecting it. On the Bluetooth 4.0 standard, the user can connect to other remote BLE devices without any pairing process. Even the user can use advertisements to send a few amount of data, as it has been described in previous sections. However, these processes are not secure.

The Bluetooth 4.0 standard uses AES-128 link layer encryption. The encryption can be used in the connection processes to make them secure. The encryption of the connection can be started using the function `encryptConnection`, providing the handler of the established connection. The handler is usually zero, unless other connections are enabled.

Example of use:

```
{
 BLE. encryptConnection(handler);
}
```

By default, Man In The Midle (MITM) is set to the "just works" mode, but the user can change this parameter with the function `setSecurityParameters`, besides than key size and I/O capabilities. More information is provided on the manufacturer documentation.

Example of use:

```
{
 // setSecurityParameters( mitm, min_key_size, io_capabilities)
 // enable MITM, set key size to 7 and IO capabilities to display only.
 BLE.setSecurityParameters(BLE_ENABLE_MITM, 7, 0);
}
```

The bonding processes are the long term storage of encryption keys used by each BLE module. To enable bonding in a BLE device, the function `setBondableMode` should be used.

Example of use:

```
{
 BLE.setBondableMode(BLE_ENABLE_BONDING);
}
```

Starting a connection with bonding will imply next steps:

1. Node 1 advertises
2. Node 2 detects advertisements of node 1
3. Node 2 starts a connection with node 1
4. Node 2 request bonding
5. Both nodes exchange keys
6. Both nodes store keys for future usage

The 6 digit key can be set using the `setKey` function. Besides that, the number of bonded devices can be listed with `getBonding` function. On the other hand, the function `deleteBonding` erases from memory all bonded devices.

Finally, there is a specific characteristic property which enables authenticated read / write operations. When this property is enabled, both devices involved in the connection needs to be bonded and the link encrypted. Refer to the module manufacturer documentation for more information.

7.4. WhiteList

The Whitelist is a list of BLE devices with special permissions. It is thought to save trusted BLE devices, allowed to establish a connection or write certain attributes. It is also used in active scans (see advertisement policy and scan policy).

A new member can be added using the `whiteListAppend` function, providing its MAC address. It also can be removed with the `whiteListRemove` function. If the user wants to clear all whitelist, the function `whiteListClear` should be used.

Example of use:

```
{
  // add new member
  BLE.whiteListAppend("001122334455");

  //remove new member
          BLE.whiteListRemove("001122334455");

          // clear all members
          BLE.whiteListClear();
}
```

This functions will not have effect while advertising, scanning or being connected, so the Whitelist should be managed in the right moment.

In the Waspmote Development section you can find a complete example about using this function.

Go to: **http://www.libelium.com/development/waspmote/examples/ble-14-managing-whitelist**

8. Connecting Waspmote to a smartphone

It is also possible to connect a smartphone (compatible with the Bluetooth 4.0 standard) with the BLE module using one of the multiple commercial applications available. With them, it is easy to scan and connect to the BLE module, and perform the basic operations on the characteristics defined in the BLE module profile.

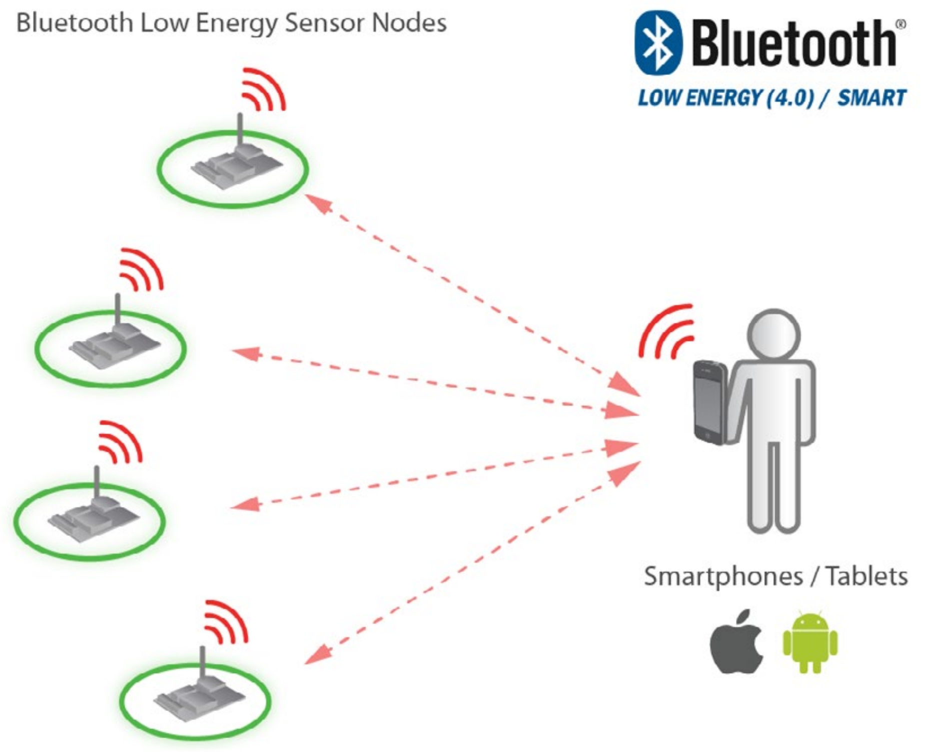


Figure : Bluetooth low energy sensor nodes detected by a smartphone.

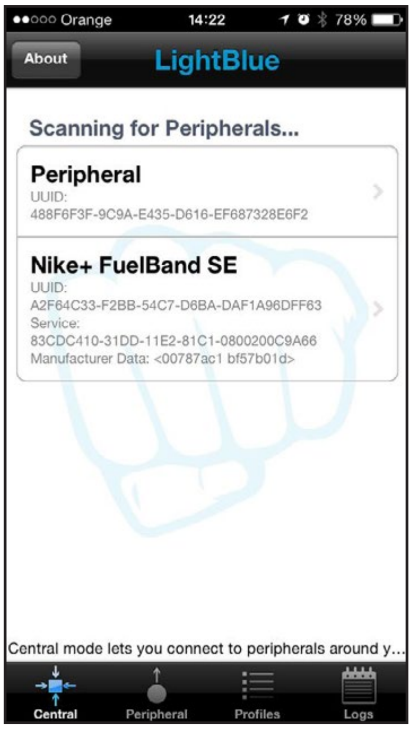
In this guide, the app "Light blue" is used for iOS, and the app "nRF Master control" is used for Android, but similar apps can be used to perform same actions. Both can be found in respective app markets.

***NOTE:** During the next sections we will see an examples of applications that can be used to get sensor data sent directly by Waspmote + BLE module to the smartphone. However, the final idea is that the Developer programs his own app so that the information can be stored and shown as desired. The next sections are shown as a quick proof of concept of the capabilities of this technology.*

Scanning for BLE devices

It is easy to scan other BLE modules by pressing the scan button. Each scanned device shows its MAC address and the RSSI. Be sure that the slave is visible and connectable by your smartphone, as described in previous sections.

iOS



●●○○○ Orange 14:22 78%

About — LightBlue

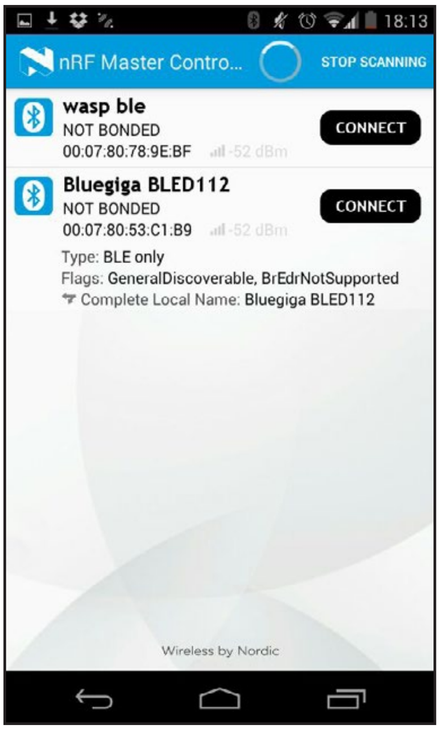
Scanning for Peripherals...

Peripheral
UUID:
488F6F3F-9C9A-E435-D616-EF687328E6F2

Nike+ FuelBand SE
UUID:
A2F64C33-F2BB-54C7-D6BA-DAF1A96DFF63
Service:
83CDC410-31DD-11E2-81C1-0800200C9A66
Manufacturer Data: <00787ac1 bf57b01d>

Central mode lets you connect to peripherals around y...

Android



18:13

nRF Master Contro... STOP SCANNING

wasp ble
NOT BONDED
00:07:80:78:9E:BF -52 dBm
CONNECT

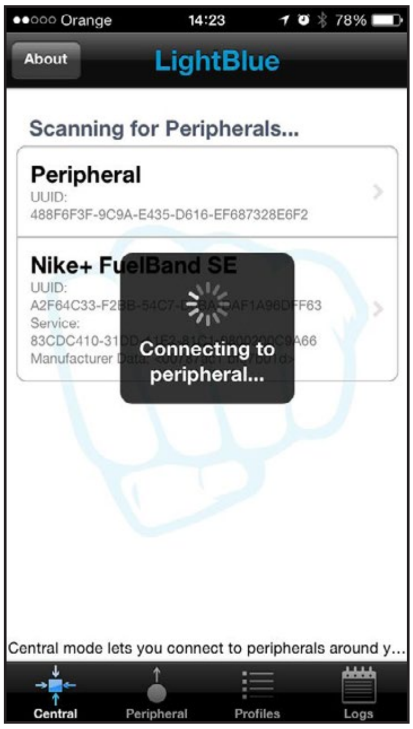
Bluegiga BLED112
NOT BONDED
00:07:80:53:C1:B9 -52 dBm
CONNECT
Type: BLE only
Flags: GeneralDiscoverable, BrEdrNotSupported
Complete Local Name: Bluegiga BLED112

Figure : Scanning for the BLE device

Connecting to a BLE device

When the desired BLE module is found, it can be connected. The smartphone will be the master and the BLE module will become the slave.

iOS



●●○○○ Orange 14:23 78%

About — LightBlue

Scanning for Peripherals...

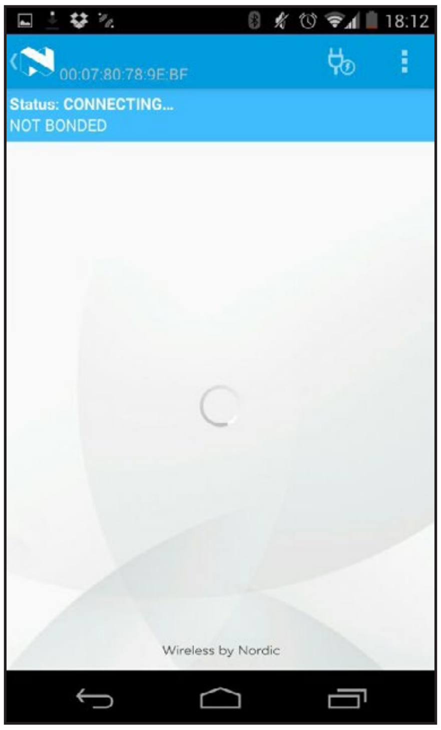
Peripheral
UUID:
488F6F3F-9C9A-E435-D616-EF687328E6F2

Nike+ FuelBand SE

Connecting to peripheral...

Central mode lets you connect to peripherals around y...

Android



18:12

00:07:80:78:9E:BF

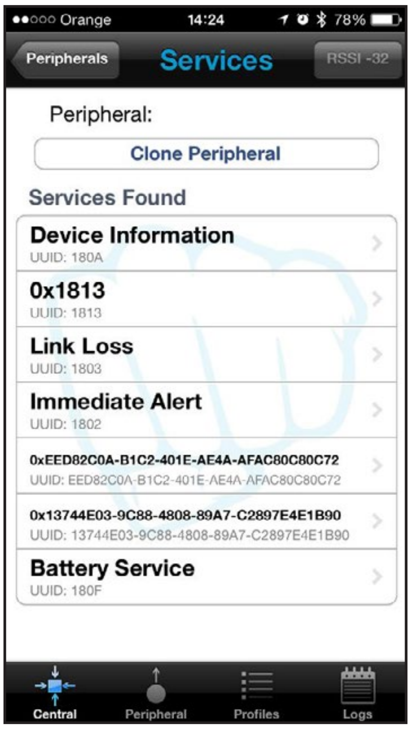
Status: CONNECTING...
NOT BONDED

Figure : Connecting to the BLE device

Receiving data from a remote BLE device

If the connection is established successfully, now the user can navigate through the profile of the slave.

iOS



●●○○○ Orange 14:24 78%

Peripherals | Services | RSSI -32

Peripheral:

Clone Peripheral

Services Found

Device Information
UUID: 180A

0x1813
UUID: 1813

Link Loss
UUID: 1803

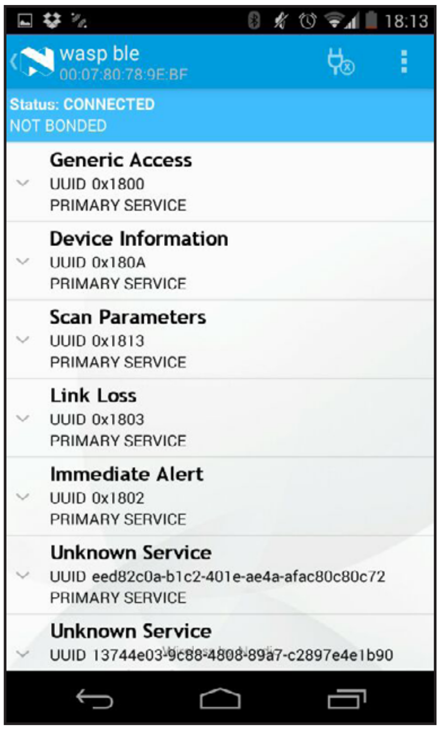
Immediate Alert
UUID: 1802

0xEED82C0A-B1C2-401E-AE4A-AFAC80C80C72
UUID: EED82C0A-B1C2-401E-AE4A-AFAC80C80C72

0x13744E03-9C88-4808-89A7-C2897E4E1B90
UUID: 13744E03-9C88-4808-89A7-C2897E4E1B90

Battery Service
UUID: 180F

Android



18:13

wasp ble
00:07:80:78:9E:BF

Status: CONNECTED
NOT BONDED

Generic Access
UUID 0x1800
PRIMARY SERVICE

Device Information
UUID 0x180A
PRIMARY SERVICE

Scan Parameters
UUID 0x1813
PRIMARY SERVICE

Link Loss
UUID 0x1803
PRIMARY SERVICE

Immediate Alert
UUID 0x1802
PRIMARY SERVICE

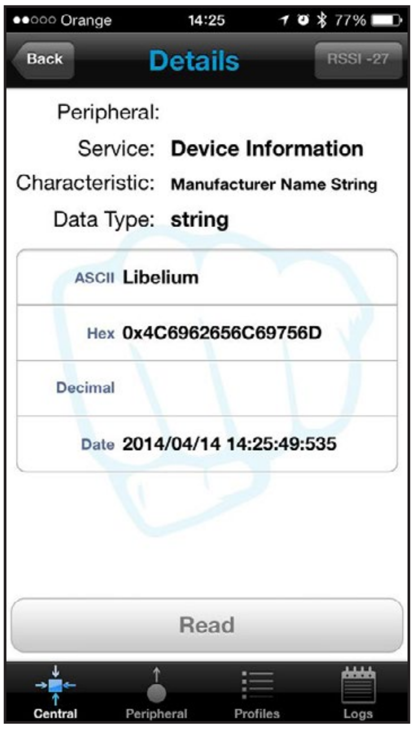
Unknown Service
UUID eed82c0a-b1c2-401e-ae4a-afac80c80c72
PRIMARY SERVICE

Unknown Service
UUID 13744e03-9c88-4808-89a7-c2897e4e1b90

Figure : Reading slave profile

If the characteristic is readable, the user can easily access to the value of the characteristic.

iOS



●●○○○ Orange 14:25 77%

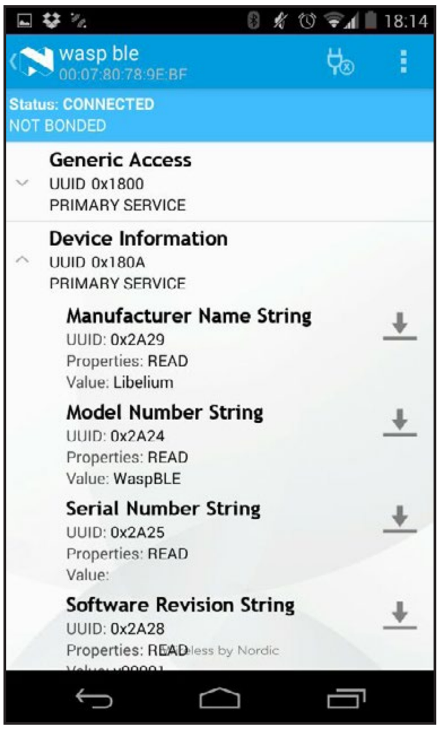
Back | Details | RSSI -27

Peripheral:
Service: Device Information
Characteristic: Manufacturer Name String
Data Type: string

ASCII Libelium
Hex 0x4C6962656C69756D
Decimal
Date 2014/04/14 14:25:49:535

Read

Android



18:14

wasp ble
00:07:80:78:9E:BF

Status: CONNECTED
NOT BONDED

Generic Access
UUID 0x1800
PRIMARY SERVICE

Device Information
UUID 0x180A
PRIMARY SERVICE

Manufacturer Name String
UUID: 0x2A29
Properties: READ
Value: Libelium

Model Number String
UUID: 0x2A24
Properties: READ
Value: WaspBLE

Serial Number String
UUID: 0x2A25
Properties: READ
Value:

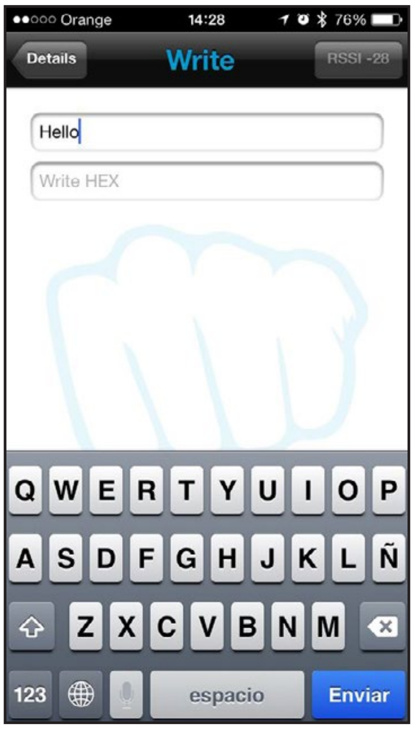
Software Revision String
UUID: 0x2A28
Properties: READ

Figure : Reading characteristic

Sending data to a remote BLE device

If the characteristic is writable, then the user can enter a new value and write it onto the characteristic.

iOS



Android

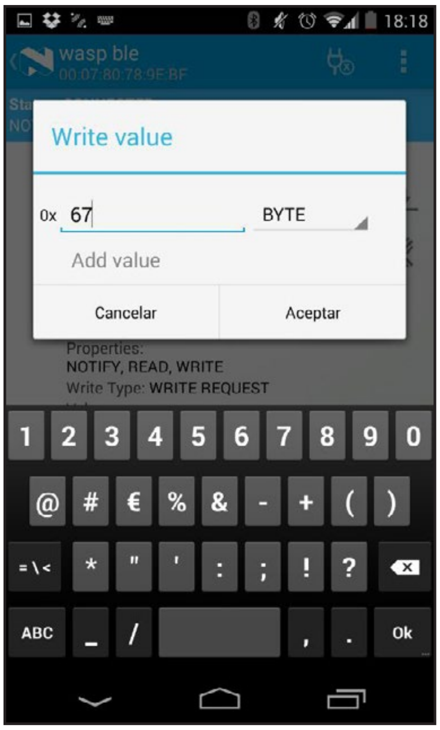
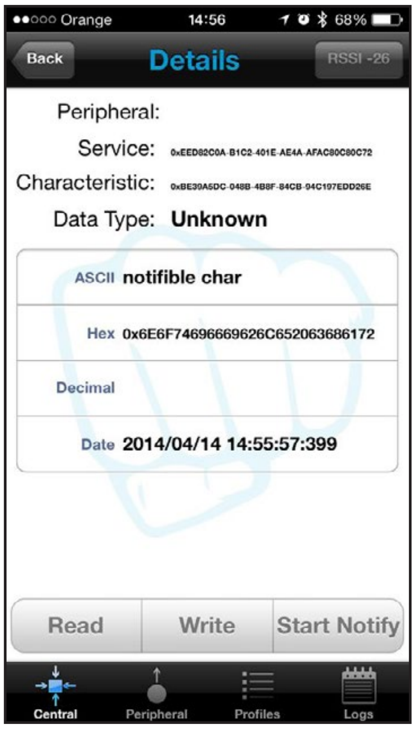


Figure : Writing a characteristic

Subscribing to a characteristic

If the characteristic is notifiable, the user can subscribe to it to know when it changes.

iOS


●●○○○ Orange 14:56 68%
Back
Details
RSSI -26

Peripheral:
Service: 0xEED82C0A-B1C2-401E-AE4A-AFAC80C80C72
Characteristic: 0xBE39A5DC-048B-4B8F-84CB-94C197EDD26E
Data Type: Unknown

ASCII notifible char
Hex 0x6E6F74696669626C652063686172
Decimal
Date 2014/04/14 14:55:57:399

Read | Write | Start Notify
Central | Peripheral | Profiles | Logs

Android

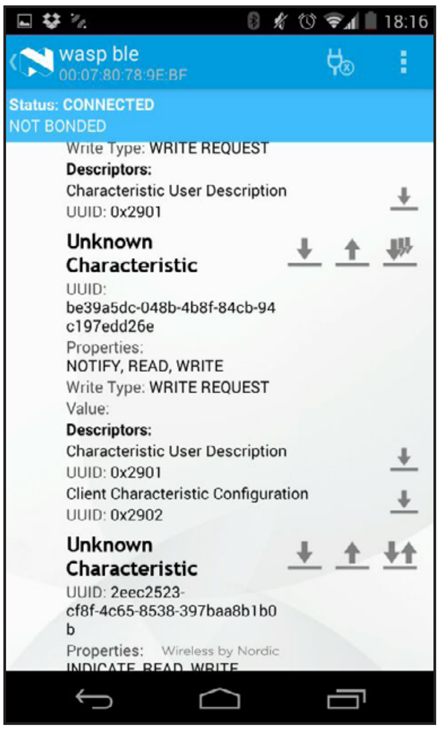
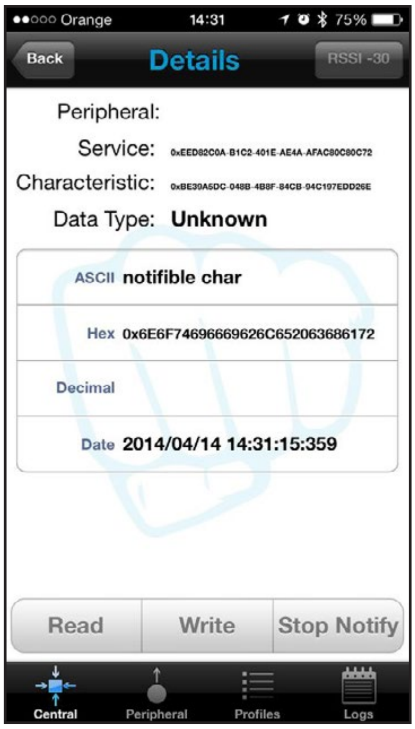

18:16
wasp ble
00:07:80:78:9E:BF
Status: CONNECTED
NOT BONDED
Write Type: WRITE REQUEST
Descriptors:
Characteristic User Description
UUID: 0x2901
Unknown Characteristic
UUID:
be39a5dc-048b-4b8f-84cb-94c197edd26e
Properties:
NOTIFY, READ, WRITE
Write Type: WRITE REQUEST
Value:
Descriptors:
Characteristic User Description
UUID: 0x2901
Client Characteristic Configuration
UUID: 0x2902
Unknown Characteristic
UUID: 2eec2523-cf8f-4c65-8538-397baa8b1b0b
Properties: Wireless by Nordic
INDICATE, READ, WRITE

Figure : Subscribing to a characteristic

Once the smartphone is subscribed, the value of the characteristic will be updated in the smartphone every time it is changed by the slave.

iOS



●●○○○ Orange 14:31 75%

Back **Details** RSSI -30

Peripheral:
Service: 0xEED82C0A-B1C2-401E-AE4A-AFAC80C80C72
Characteristic: 0xBE30A5DC-048B-4B8F-84CB-94C197EDD26E
Data Type: **Unknown**

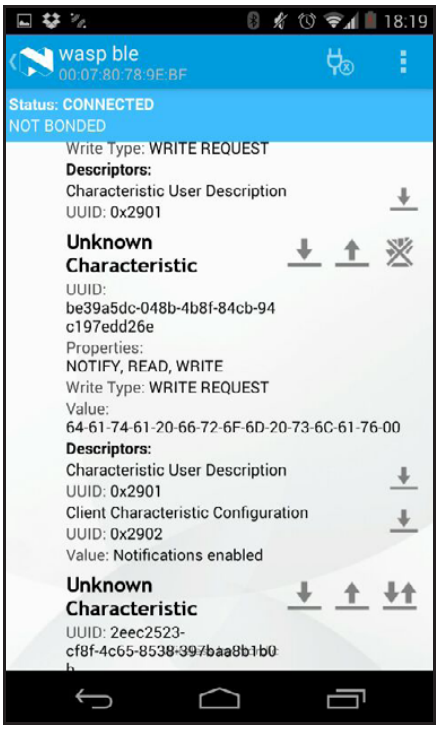
ASCII **notifible char**

Hex **0x6E6F74696669626C652063686172**

Decimal

Date **2014/04/14 14:31:15:359**

Android



18:19

wasp ble
00:07:80:78:9E:BF

Status: CONNECTED
NOT BONDED

Write Type: WRITE REQUEST
Descriptors:
Characteristic User Description
UUID: 0x2901

Unknown Characteristic
UUID:
be39a5dc-048b-4b8f-84cb-94
c197edd26e
Properties:
NOTIFY, READ, WRITE
Write Type: WRITE REQUEST
Value:
64-61-74-61-20-66-72-6F-6D-20-73-6C-61-76-00
Descriptors:
Characteristic User Description
UUID: 0x2901
Client Characteristic Configuration
UUID: 0x2902
Value: Notifications enabled

Unknown Characteristic
UUID: 2eec2523-
cf8f-4c65-8538-397baa8b1b0

Figure : Notified characteristic

9. Default Profile on BLE module

The internal microcontroller of the Bluetooth low energy module has a default profile containing the specific configuration to use this module with Waspmote. Besides than the necessary hardware configuration, there is a default profile with the services and characteristics (with attributes like descriptors, values, etc) to be read /written by other BLE modules, according to the Bluetooth standard. If the user needs more information about how the information is set into a BLE profile, please refer to the Bluetooth SIG website **https://www.bluetooth.org**.

The default profile is designed to allow sending /receiving information to /with other BLE devices, including many value types and diferent characteristics, as is described below.

Service	Characteristic	UUID	Properties
Generic Access Profile UUID: 2A00	Device Name	2A00	Read / Write
	Device appearance	2A01	Read
Device information UUID: 180A	Manufacturer Name	2A29	Read
	Module	2A24	Read
	Serial Number	2A25	Read
	Software version	2A28	Read
Scan parameters UUID: 1813	Scan interval window	2A4F	Write_no_response
	Scan refresh	2A31	Notify
Link loss UUID: 1803	Alert level	2A06	Read / Write
Immediate alert UUID: 1802	Alert level	2A06	Write_no_response
User Serivice 1 UUID: eed82c0a-b1c2-401e-ae4a-afac80c80c72	1.0	985dc5e5-1d8c-405f-b50f-d29f700e13a1	Read / Write
	1.1	e1987994-6773-4196-94ee-3dc0474a74e2	Read / Write
	1.2	99946582-4ca1-43b3-aa79-020bf8fa0bff	Read / Write
	1.3	88017ef1-515a-4029-b27b-6eb77f82e811	Read / Write
	1.4	be39a5dc-048b-4b8f-84cb-94c197edd26e	Read / Write / Notify
	1.5	2eec2523-cf8f-4c65-8538-397baa8b1b0b	Read / Write / Indicate
	1.6	a5853c93-08af-4186-8dba-b4c0cc74a23b	Read / Write / Notify
	1.7	a94ba516-c627-4e00-a28b-b5cd825d8e14	Read / Write / Indicate
	1.8	013dc1df-9b8c-4b5c-949b-262543eba78a	Read / Write
	1.9	e562b410-e8d2-4fe0-89c1-91432f108fe1	Read / Notify
	1.10	acfe574f-ceaf-4290-90d4-4a94b0c5b8ef	Read / Indicate
	1.11	fe355d2b-d02c-477f-b397-9a728800de11	Write / Notify
	1.12	362ba79d-b620-41d3-89ee-48f865559129	Write / Indicate
	1.13	3b54d144-a5f4-4444-aabc-7ada95be9498	Write_no_response
	1.14	ade7a273-89f9-49e1-b9d4-3cb36bce261b	Write
	1.15	5b552788-5c7b-4ce8-8362-cf5dd093251d	Read / Notify / indicate
User service 2 UUID: 13744e03-9c88-4808-89a7-c2897e4e1b90	2.0	b51b827c-98c6-4e1c-a567-e8f48faa7e76	Read / Write / Auth Write
	2.1	9737456d-d754-460d-a22b-39ab73ccd0af	Read / Auth Write / notify
	2.2	d161afd6-f02f-49a4-8c14-5349a6de08d7	Read / Auth Write / Indicate
	2.3	9f1c0185-57c6-4037-a4e8-9f5d3b3c0ca6	Auth read / write

Figure : Default profile description

10. Other API functions

There are other less relevant functions used for secondary features or specific features. All these functions are described below.

Function	Description
`GetOwnMac`	Returns the BLE module MAC address.
`GetTemp`	Reads the internal Temperature sensor of the BLE module. Not calibrated. +-10ªC accuracy.
`sayHello`	Just checks comunication with the BLE module.
`waitEvent`	By default, the module does not parse events automatically, so if the user send commands which can produce events, they will have to be catched manually.
`parseEvent`	Identifies a received event.

Besides than the already developed functions, remember that there are some functions available to send custom commands directly to the module. The user is free to develop his own function.

Finally, there can be applications that may involve human signal management, like heart rate monitors, etc. It is important to remark that the Libelium BLE module is not authorized for use as critical component in life support devices or systems, so its usage involving these applications is limited for testing/demonstration purposes.

11. Code examples and extended information

In the Waspmote Development section you can find complete examples:

http://www.libelium.com/development/waspmote/examples

Example:

```
/*
 *  ------------------ [BLE_01] - Normal scan -------------------
 *
 *  Explanation: This example shows how to make a normal scan with
 *  Bluetooth low energy, printing number of discovered devices
 *  and scan results stored in EEPROM.
 *
 *  Copyright (C) 2014 Libelium Comunicaciones Distribuidas S.L.
 *  http://www.libelium.com
 *
 *  This program is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 *  Version:          0.1
 *  Design:           David Gascón
 *  Implementation:   Javier Siscart
 */

#include <WaspBLE.h>

void setup()
{

  USB.println(“BLE_01 Example”);

  uint8_t flag = BLE.ON(SOCKET0);
  // 0. Turn BLE module ON
  if (flag == 1)
  {
    USB.println(“BLE is ON.”);
  }
  else
  {
     USB.println(“BLE not initialized.”);
     USB.println(flag,DEC);
  }
}

void loop()
{
  // 1. Normal scan
  USB.println(“New scan for 5s..”);
  BLE.scanNetwork(5);
  USB.printf(“discovered devices: %d\r\n”, BLE.numberOfDevices);

  USB.println(“Printing Last inquiry saved on EEPROM:”);
  uint8_t dummy = BLE.printInquiry();
  USB.printf(“devices printed:%d\r\n\r\n”,dummy);
}
```

12. API changelog

Function	Changelog	Version
`WaspBLE::sendCommand`	Improved	v009 → v010
`WaspBLE::readEvent`	Removed	v009 → v010
`uint8_t WaspBLE::reset()`	Improved	v009 → v010
`uint16_t WaspBLE::setAdvData()`	New prototype	v009 → v010
`uint16_t WaspBLE::whiteListAppend()`	Improved	v009 → v010
`uint16_t WaspBLE::whiteListRemove()`	Improved	v009 → v010
`uint16_t WaspBLE::disconnect()`	Improved	v009 → v010
`uint8_t WaspBLE::wakeUp()`	Improved	v009 → v010
`uint8_t WaspBLE::waitEvent()`	Improved	v009 → v010
`uint16_t WaspBLE::writeLocalAttribute()`	New prototypes	v009 → v010
`uint16_t WaspBLE::writeLocalAttribute()`	Improved	v009 → v010
`uint16_t WaspBLE::attributeRead()`	Improved	v009 → v010
`uint16_t WaspBLE::attributeWrite()`	New prototype	v009 → v010
`uint8_t WaspBLE::parseEvent()`	Created	v009 → v010
`uint8_t WaspBLE::getStatus()`	Created	v009 → v010
`Event list`	Created	v009 → v010
`uint8_t eventLength`	Deleted	v009 → v010
`uint8_t i`	Changed to uint16_t	v009 → v010

13. Document changelog

Form V4.0 to V4.1:

- Api changelog modified
- Example links modified