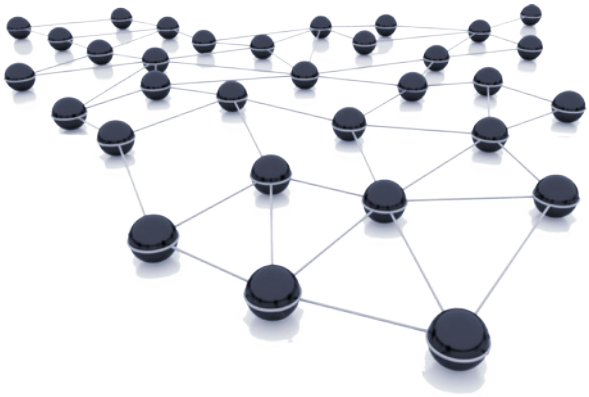
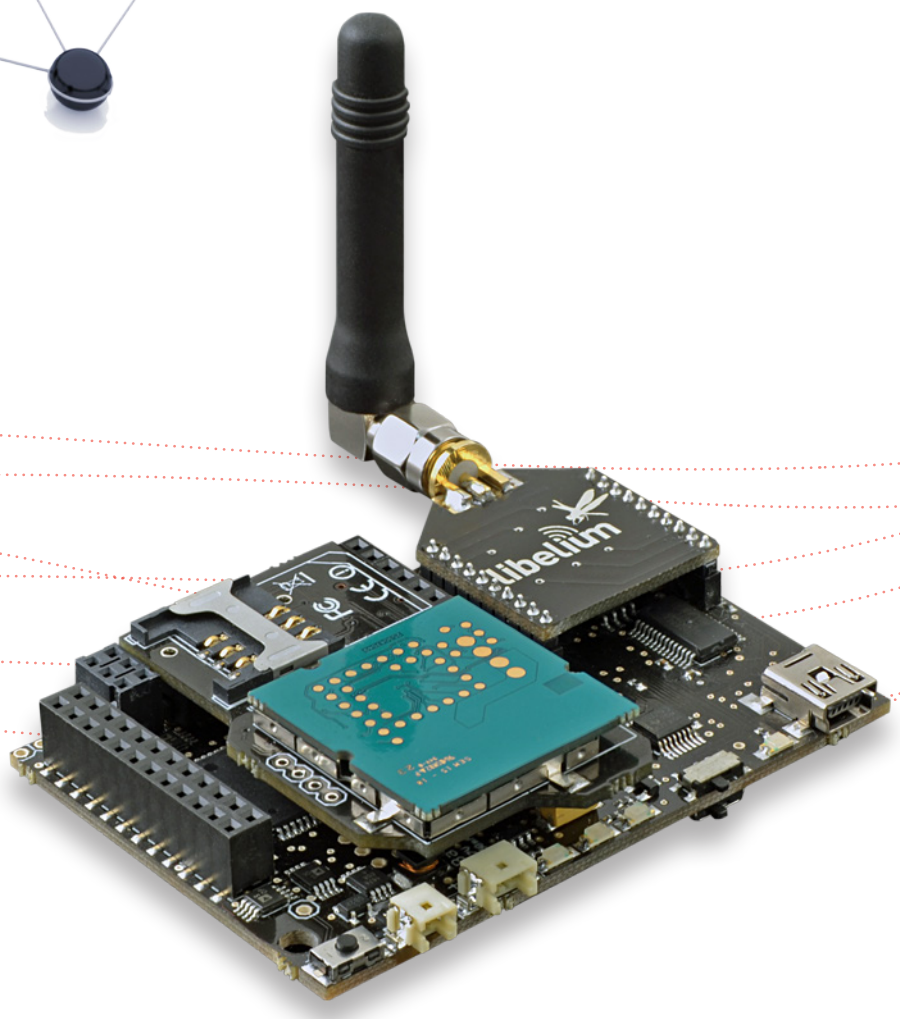
# Waspmote 802.15.4

## Networking Guide






libelium

wasp mote

Document Version: v4.3 - 02/2014


# INDEX

# 1. Hardware

| Module | Frequency | TX power | Sensitivity | Channels | Distance |
|---|---|---|---|---|---|
| PRO | 2,405 – 2,465GHz | 63.1mW | -100dBm | 12 | 7000m |


*Figure 1: XBee 802.15.4 PRO*

The frequency used is the free band of 2.4GHz, using 12 channels with a bandwidth of 5MHz per channel.

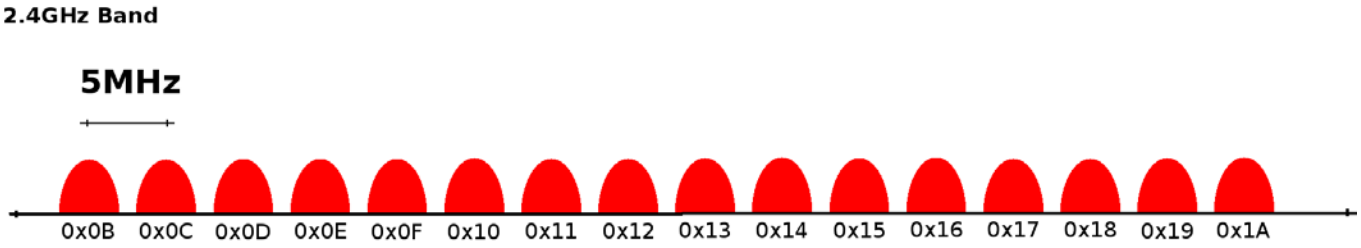



*Figure 2: Frequency channels in the 2.4GHz band*

***Note:*** *From February 2014, Libelium no longer offers the XBee 802.15.4 wireless module in the Standard or Normal version. From that date, XBee 802.15.4 is only offered in the PRO version. Standard version only differs in lower output power levels and worse sensitivity. On the other hand, Standard version can radiate in a wider number of channels (16). Basically, both versions work in the same way, so this guide is still useful for Standard version users.*

| Channel Number | Frequency | Supported by |
|---|---|---|
| 0x0C – Channel 12 | 2,405 – 2,410 GHz | PRO |
| 0x0D – Channel 13 | 2,410 – 2,415 GHz | PRO |
| 0x0E – Channel 14 | 2,415 – 2,420 GHz | PRO |
| 0x0F – Channel 15 | 2,420 – 2,425 GHz | PRO |
| 0x10 – Channel 16 | 2,425 – 2,430 GHz | PRO |
| 0x11 – Channel 17 | 2,430 – 2,435 GHz | PRO |
| 0x12 – Channel 18 | 2,435 – 2,440 GHz | PRO |
| 0x13 – Channel 19 | 2,440 – 2,445 GHz | PRO |
| 0x14 – Channel 20 | 2,445 – 2,450 GHz | PRO |
| 0x15 – Channel 21 | 2,450 – 2,455 GHz | PRO |
| 0x16 – Channel 22 | 2,455 – 2,460 GHz | PRO |
| 0x17 – Channel 23 | 2,460 – 2,465 GHz | PRO |

*Figure 3: Channels used by the XBee modules in 2.4GHz*

The XBee 802.15.4 modules comply with the standard **IEEE 802.15.4** which defines the physical level and the link level (MAC layer). The XBee modules add certain functionalities to those contributed by the standard, such as:

- **Node discovery:** certain information has been added to the packet headers so that they can discover other nodes on the same network. It allows a node discovery message to be sent, so that the rest of the network nodes respond indicating their data (Node Identifier, @MAC, @16 bits, RSSI).
- **Duplicated packet detection:** This functionality is not set out in the standard and is added by the XBee modules.

With a view to obtain frames totally compatible with the IEEE802.15.4 standard and enabling inter-operability with other chipsets, the **xbee802.setMacMode(m)** command has been created to select at any time if the modules are to use a totally compatible heading format, or conversely enable the use of extra options for node discovery and duplicated packets detection.

Encryption is provided through the **AES 128b** algorithm. Specifically through the **AES-CTR type.** In this case the Frame Counter field has a unique ID and encrypts all the information contained in the **Payload** field which is the place in the 802.15.4 frame where data to be sent is stored.

The way in which the libraries have been developed for the module programming makes encryption activation as simple as running the initialization function and giving it a key to use in the encryption process.

```
{
  xbee802.setEncryptionMode(1);
  xbee802.setLinkKey(key);
}
```

Extra information about the encryption systems in 802.15.4 and ZigBee sensor networks can be accessed in the Development section of the Libelium website, specifically in the document: "Security in 802.15.4 and ZigBee networks"

The classic layout of this type of network is Star topology, as the nodes establish point to point connections with brother nodes through the use of parameters such as the MAC or network address.

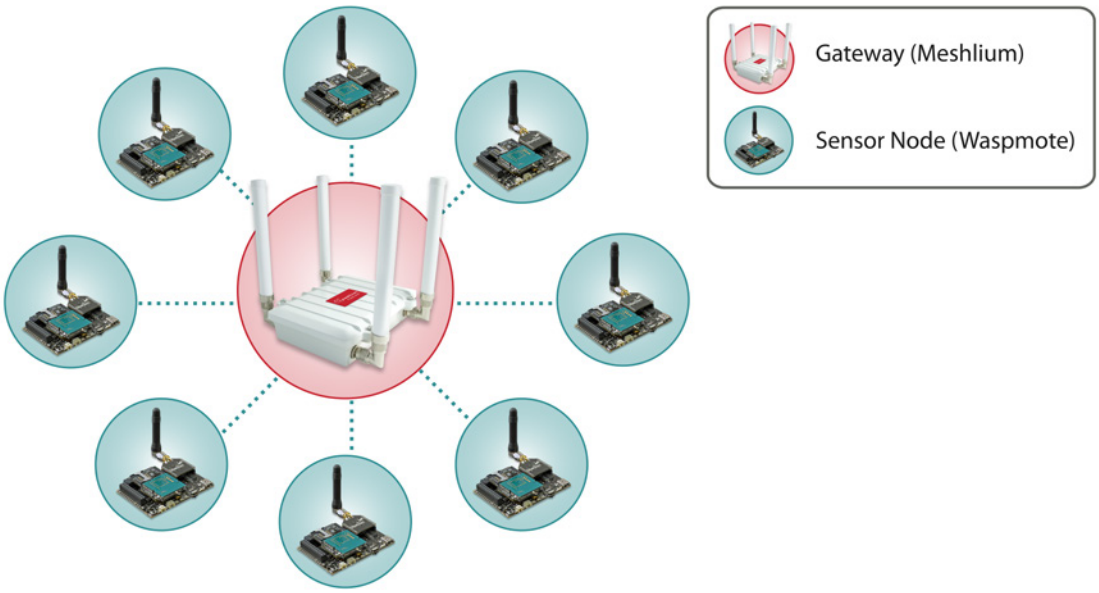



*Figure 4: Star topology*

Regarding the energy section, the transmission power can be adjusted to several values:

| Parameter | Tx XBee-PRO |
|---|---|
| 0 | 10dBm |
| 1 | 12dBm |
| 2 | 14dBm |
| 3 | 16dBm |
| 4 | 18dBm |

*Figure 5: Transmission power values*

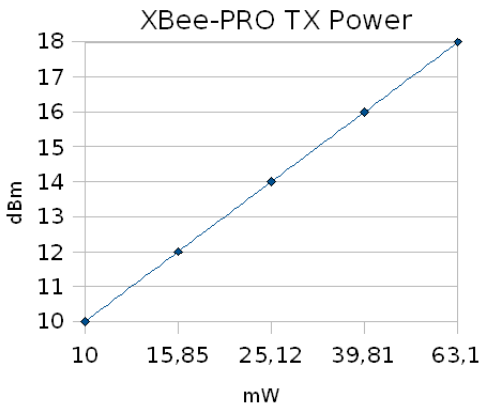



*Figure 6: XBee- PRO TX Power*

# 2. General Considerations

## 2.1. Waspmote Libraries

### 2.1.1. Waspmote XBee Files

WaspXBeeCore.h, WaspXBeeCore.cpp, WaspXBee802.h, WaspXBee802.cpp

It is mandatory to include the XBee802 library when using this module. The following line must be introduced at the beginning of the code:

```
#include <WaspXBee802.h>
```

### 2.1.2. Constructor

To start using the Waspmote XBee library, an object from class 'WaspXBee802' must be created. This object, called 'xbee802', is created inside the Waspmote XBee library and it is public to all libraries. It is used through the guide to show how the Waspmote XBee library works.

When creating this constructor, some variables are defined with a value by default.

## 2.2. API Functions

Through the guide there are many examples of using parameters. 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

```
{
  xbee802.getOwnMacLow(); // Get 32 lower bits of MAC Address
  xbee802.getOwnMacHigh(); // Get 32 upper bits of MAC Address
}
```

Related Variables

xbee802.sourceMacHigh[0-3] ⟶ stores the 32 upper bits of MAC address
xbee802.sourceMacLow [0-3] ⟶ stores the 32 lower bits of MAC address

When returning from 'xbee802.getOwnMacLow' the related variable 'xbee802.sourceMacLow' will be filled with the appropriate values. Before calling the function, the related variable is created but it is empty or with a default value.

There are three error flags that are filled when the function is executed:

- error_AT: it stores if some error occurred during the execution of an AT command function
- error_RX: it stores if some error occurred during the reception of a packet
- error_TX: it stores if some error occurred during the transmission of a packet

All the functions also return a **flag** to know if the function called was successful or not. Available values for this flag:

- 0 : Success. The function was executed without errors and the exposed variable was filled.
- 1 : Error. The function was executed but an error occurred while executing.
- 2 : Not executed. An error occurred before executing the function.
- -1 : Function not allowed in this module.

To store parameter changes after power cycles, it is needed to execute the writeValues function.

Example of use

```
{
  xbee802.writeValues(); // Keep values after power down

}
```

## 2.3. API extension

All the relevant and useful functions have been included in the Waspmote API, although any XBee command can be sent directly to the transceiver.

Example of use

```
{
  xbee802.sendCommandAT("CH#"); // Executes command ATCH
}
```

Related Variables

xbee802.commandAT[0-100] ⟶ stores the response given by the module up to 100 bytes

• Sending AT commands example:

**http://www.libelium.com/development/waspmote/examples/802-12-send-atcommand**

## 2.4. Waspmote reboot

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

## 2.5. Constants pre-defined

There are some constants pre-defined in a file called 'WaspXBeeCore.h'. These constants define some parameters like the maximum data size. The most important constants are explained next:

- MAX_DATA: (default value is 100) it defines the maximum available data size for a packet. This constant must be equal or bigger than the data is sent on each packet. This size shouldn't be bigger than 1500.
- MAX_PARSE: (default value is 300) it defines the maximum data that is parsed in each call to 'treatData'. If more data are received, they will be stored in the UART buffer until the next call to 'treatData'. However, if the UART buffer is full, the following data will be written on the buffer, so be careful with this matter.
- MAX_BROTHERS: (default value is 5) it defines the maximum number of brothers that can be stored.
- MAX_FINISH_PACKETS: (default value is 5) it defines the maximum number of finished packets that can be stored.
- XBEE_LIFO: it specifies the LIFO replacement policy. If one packet is received and the finished packets array is full, this policy will free the previous packet received and it will store the data from the last packet.
- XBEE_FIFO: it specifies the FIFO replacement policy. If one packet is received and the finished packets array is full, this policy will free the first packet received and it will store the data from the last packet.
- XBEE_OUT: it specifies the OUT replacement policy. If one packet is received and the finished packets array is full, this policy will discard this packet.

# 3. Initialization

Before starting to use a module, it needs to be initialized. During this process, the UART to communicate with the module has to be opened and the XBee switch has to be set on.

## 3.1. Setting ON

It initializes all the global variables, opens the correspondent UART and switches the XBee ON. The baud rate used to open the UART is defined on the library (115200bps by default).

It returns nothing.

The initialized variables are:

- **protocol:** specifies the protocol used (802.15.4 in this case).
- **pos:** specifies the position to use in received packets.
- **discoveryOptions:** specifies the options in Node Discovery.
- **awakeTime:** specifies the time to be awake before go sleeping.
- **sleepTime:** specifies the time to be sleeping.
- **scanChannels:** specifies the channels to scan.
- **scanTime:** specifies the time to scan each channel.
- **encryptMode:** specifies if encryption mode is enabled.
- **powerLevel:** specifies the power transmission level.
- **timeRSSI:** specifies the time RSSI LEDs are on.
- **sleepOptions:** specifies the options for sleeping.
- **retries:** specifies the number of retries to execute in addition to the three retries defined in the 802.15.4 protocol.
- **delaySlots:** specifies the minimum value of the back-off exponent in CSMA/CA.
- **macMode:** specifies the Mac Mode used.
- **energyThreshold:** specifies the energy threshold used to determine if the channel is free.
- **counterCCA:** specifies the number of times too much energy was found in the channel.
- **counterACK:** specifies the number of times an ACK was lost.

Example of use

```
{
  xbee802.ON(); // Opens the UART0 by default and switches the XBee ON
}
```

**Expansion Radio Board (XBee 802.15.4)**

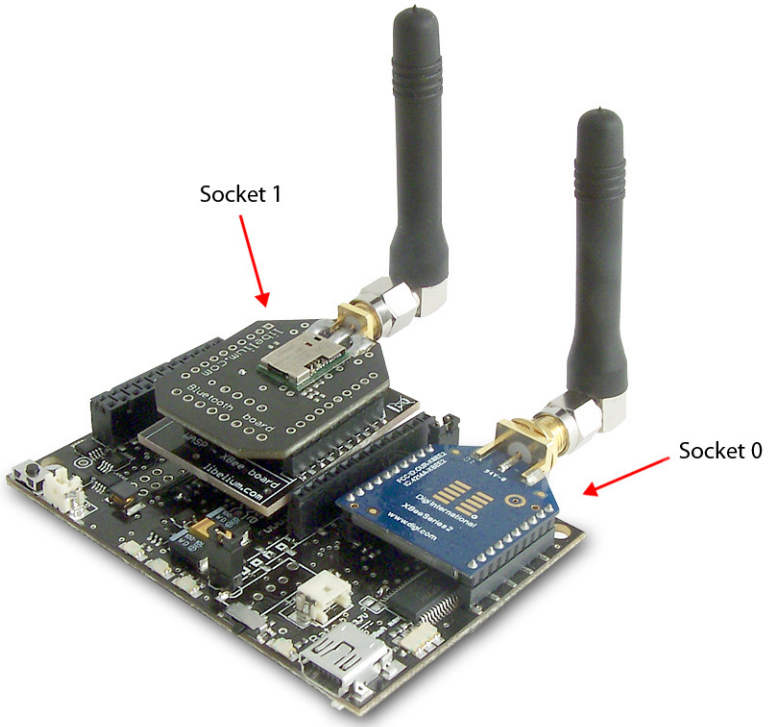
The new Expansion Board allows to connect two radios at the same time in the Waspmote sensor platform. This means a lot of different combinations are now possible using any of the nine radios available for Waspmote: 802.15.4, ZigBee, Bluetooth, RFID, Wifi, GPRS, 3G/GPRS, 868 MHz and 900 MHz.

Some of the possible combinations are:

- ZigBee - Bluetooth
- ZigBee - RFID
- ZigBee - Wifi
- ZigBee - GPRS
- Bluetooth - RFID
- RFID - GPRS
- etc.

***Remark**: the GPRS module and 3G/GPRS module do not need the Expansion Board to be connected to Waspmote. It can be plugged directly in the GPRS socket.*

In the next photo you can see the sockets available along with the UART assigned. On one hand, SOCKET0 allows to plug any kind of radio module through the UART0. On the other hand, SOCKET1 permits to connect a radio module through the UART1.




The API provides a function called 'ON' in order to switch the XBee module on. This function supports a parameter which permits to select the SOCKET. It is possible to choose between SOCKET0 and SOCKET1.

Selecting SOCKET0 (both are valid):

```
xbee802.ON();
xbee802.ON(SOCKET0);
```

Selecting SOCKET1:

```
xbee802.ON(SOCKET1);
```

In the case two XBee-802.15.4 modules are needed (each one in each socket), it will be necessary to create a new object from WaspXBee802 class. By default, there is already an object called 'xbee802' normally used for regular SOCKET0.

In order to create a new object it is necessary to put the following declaration in your Waspmote code:

```
WaspXBee802 xbee802_2 = WaspXBee802();
```

Finally, it is necessary to initialize both modules. For example, xbee802 is initialized in SOCKET0 and xbee802_2 in SOCKET1 as follows:

```
xbee802.ON(SOCKET0);
xbee802_2.ON(SOCKET1);
```

The rest of functions are used the same way as they are used with older API versions. In order to understand them we recommend to read this guide.

WARNING:

- Avoid to use DIGITAL7 pin when working with Expansion Board. This pin is used for setting the XBee into sleep.
- Avoid to use DIGITAL6 pin when working with Expansion Board. This pin is used as power supply for the Expansion Board.
- Incompatibility with Sensor Boards:
  - Gases Board: Incompatible with SOCKET4 and $NO_2$/$O_3$ sensor.
  - Agriculture Board: Incompatible with Sensirion and the atmospheric pressure sensor.
  - Smart Metering Board: Incompatible with SOCKET11 and SOCKET13.
  - Smart Cities Board: Incompatible with microphone and the CLK of the interruption shift register.
  - Events Board: Incompatible with interruption shift register.

## 3.2. Setting OFF

It closes the UART and switches the XBee OFF.

Example of use

```
{
  xbee802.OFF(); // Closes the UART and switches the XBee OFF
}
```

# 4. Node Parameters

When configuring a node, it is necessary to set some parameters which will be used lately in the network, and some parameters necessary for using the API functions.

## 4.1. MAC Address

A 64-bit RF module's unique IEEE address. It is divided in two groups of 32 bits (High and Low).

It identifies uniquely a node inside a network due to it can not be modified and it is given by the manufacturer. It is used in 64-bit unicast transmissions.

Example of use

```
{
  xbee802.getOwnMacLow(); // Get 32 lower bits of MAC Address
  xbee802.getOwnMacHigh(); // Get 32 upper bits of MAC Address
}
```

Related Variables

xbee802.sourceMacHigh[0-3]⟶ stores the 32 upper bits of MAC address

xbee802.sourceMacLow [0-3]⟶ stores the 32 lower bits of MAC address

## 4.2. Network Address

A 16-bit Network Address. It identifies a node inside a network, it can be modified at running time. It is used to send data to a node in 16-bit unicast transmissions.

Example of use

```
{
  xbee802.setOwnNetAddress(0x12,0x34); // Set 0x1234 as Network Address
  xbee802.getOwnNetAddress(); // Get Network Address
}
```

Related Variables

xbee802.sourceNA[0-1] ⟶ stores the 16-bit network address

## 4.3. PAN ID

16-bit number that identifies the network. It must be unique to differentiate a network. All the nodes in the same network should have the same PAN ID.

Example of use

```
{
  uint8_t  panid[2]={0x33,0x31}; // array containing the PAN ID
 xbee802.setPAN(panid); // Set PANID
 xbee802.getPAN(); // Get PANID
}
```

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

Related Variables

xbee802.PAN_ID[0-7] ⟶ stores the 16-bit PAN ID. It is stored in the two first positions.

# 4.4. Node Identifier

It is an ASCII string of 20 character at most which identifies the node in a network. It is used to identify a node in the application level. It is also used to search a node using its NI.

Example of use

```
{
  xbee802.setNodeIdentifier("forestnode-01"); // Set 'forestnode-01' as NI
  xbee802.getNodeIdentifier(); // Get NI
}
```

Related Variables

xbee802.nodeID[0-19] ⟶ stores the 20-byte max string Node Identifier

# 4.5. Channel

This parameter defines the frequency channel used by the module to transmit and receive.

802.15.4 defines 12 channels to be used.

- 2.405-2.465GHz : 12 channels

This module works in 2.4GHz band, having 12 channels with a 5MHz bandwidth per channel.

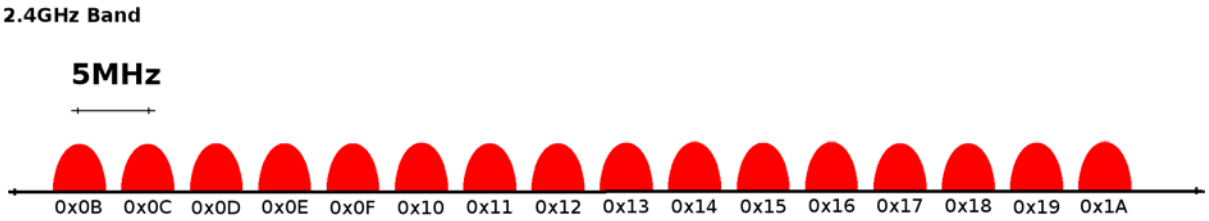



*Figure 7: Operating Frequency Bands*

| Channel Number | Frequency | Supported by |
|---|---|---|
| 0x0C – Channel 12 | 2,405 – 2,410 GHz | PRO |
| 0x0D – Channel 13 | 2,410 – 2,415 GHz | PRO |
| 0x0E – Channel 14 | 2,415 – 2,420 GHz | PRO |
| 0x0F – Channel 15 | 2,420 – 2,425 GHz | PRO |
| 0x10 – Channel 16 | 2,425 – 2,430 GHz | PRO |
| 0x11 – Channel 17 | 2,430 – 2,435 GHz | PRO |
| 0x12 – Channel 18 | 2,435 – 2,440 GHz | PRO |
| 0x13 – Channel 19 | 2,440 – 2,445 GHz | PRO |
| 0x14 – Channel 20 | 2,445 – 2,450 GHz | PRO |
| 0x15 – Channel 21 | 2,450 – 2,455 GHz | PRO |
| 0x16 – Channel 22 | 2,455 – 2,460 GHz | PRO |
| 0x17 – Channel 23 | 2,460 – 2,465 GHz | PRO |

*Figure 8: Channel Frequency Numbers*

Example of use

```
{
  xbee802.setChannel(0x0D); // Set channel
  xbee802.getChannel(); // Get Channel
}
```

Related Variables

xbee802.channel ⟶ stores the operating channel

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

# 5. Packet Parameters

## 5.1. Structure used in packets

Packets are structured in WaspXBeeCore.h using a defined structure called 'packetXBee'. This structure has many fields to be filled by the user or the application:

```
/************ IN ***********/
    uint8_t macDL[4];       // 32b Lower Mac Destination
    uint8_t macDH[4];       // 32b Higher Mac Destination
    uint8_t mode;           // 0=UNICAST ; 1=BROADCAST
    uint8_t address_type;   // 0=16b; 1=64b
    uint8_t naD[2];         // 16b Network Address Destination
    char data[MAX_DATA]; // Data of the sent message. All the data here, even when > Payload
    uint16_t data_length;  // Data sent length. Real used size of vector data[MAX_DATA]
    uint8_t SD;             // Source Endpoint
    uint8_t DE;             // Destination Endpoint
    uint8_t CID[2];         // Cluster Identifier
    uint8_t PID[2];         // Profile Identifier
    uint8_t MY_known;       // 0=unknown net address ; 1=known net address
    uint8_t opt;

 /******** APLICATION *******/
    uint8_t macSL[4];       // 32b Lower Mac Source
    uint8_t macSH[4];       // 32b Higher Mac Source
    uint8_t naS[2];         // 16b Network Address Source
    uint8_t RSSI;           // Receive Signal Strength Indicator
    uint8_t address_typeS; // 0=16b ; 1=64b
    uint8_t time;           // Specifies the time when the first fragment  was received

 /******** OUT **************/
    uint8_t deliv_status;   // Delivery Status
    uint8_t discov_status;  // Discovery Status
    uint8_t true_naD[2];    // Network Address the packet has been really sent to
    uint8_t retries;        // Retries needed to send the packet
```

**• mode**

Transmission mode chosen to transmit that packet. Available values :

0 : UNICAST transmission
1 : BROADCAST transmission

**• address_type**

Sent packet parameter: Destination address type chosen, between 16-bit and 64-bit addresses. This parameter is set by setDestinationParams function.

- 0 : MY_TYPE (16-bit Network Address)
- 1 : MAC_TYPE (64-bit MAC Address)

**• macDL & macDH**

Sent packet parameter: A 64-bit destination address. It is used to specify the MAC address of the destination node. It is used in 64-bit unicast transmissions. This parameter is used when MAC_TYPE is set as address_type.

• **naD**

Sent packet parameter: A 16-bit Destination Network Address. It is used in 16-bit unicast transmissions. This parameter is used when MY_TYPE is set as address_type.

• **data**

Data to send in the packet. It is used to store the data to send in unicast or broadcast transmissions to other nodes. All the data to send must be stored in this field. Its maximum size is defined by 'MAX_DATA', a constant defined in API libraries.

• **data_length**

Data really sent in the packet. Due to 'data' field is an array of max defined size, each packet could have a different size.

• **SD**

Not used in 802.15.4.

• **DE**

Not used in 802.15.4.

• **CID**

Not used in 802.15.4.

• **PID**

Not used in 802.15.4.

• **MY_known**

Not used in 802.15.4.

• **opt**

Options field of the received packet. Options available in 802.15.4 protocol:

- bit 0 [reserved]

- bit 1 = Address broadcast

- bit 2 = PAN broadcast

- bits 3-7 [reserved]

• **macSL & macSH**

Received packet parameter: A 64-bit Source MAC Address. It specifies the address of the node which has originally sent the message. This information is included within the header of the received XBee frame.

• **naS**

Received packet parameter: A 16-bit Source Network Address. This address corresponds to the MY parameter set in the source module by setOwnNetAddress function. This information is included within the header of the received XBee frame.

• **RSSI**

Received packet parameter: Received Signal Strength Indicator. It specifies in dBm the RSSI of the last received packet via RF.

• **address_typeS**

Received packet parameter: Address type used to send the packet by the transmitter. This addressing type matches to the one set by the transmitter using setDestinationParams function.

- 0 : MY_TYPE (16-bit transmission )
- 1 : MAC_TYPE (64-bit transmission )

• **time**

Specifies the time when the packet was received.

• **deliv_status**

Not used in 802.15.4.

• **discov_status**

Not used in 802.15.4.

• **true_naD**

Not used in 802.15.4.

• **retries**

Not used in 802.15.4.

## 5.2. Maximum payloads

Depends on the way of transmission, a maximum data payload is defined:

| | @16bit Unicast | @64bit Unicast | Broadcast |
|---|---|---|---|
| **Encrypted** | 98Bytes | 94Bytes | 95Bytes |
| **Un-Encrypted** | 100Bytes | 100Bytes | 100Bytes |

*Figure 9: Maximum Payload Size*

## 5.3. 802.15.4 Compliance

The Waspmote XBee transceiver is IEEE 802.15.4 compliance when the 'Mac Mode' is set to a value = 2. For other values, this parameter adds an extra header which let some new features to 802.15.4 protocol.

These features are:

- Node Discover: It performs a scan in the network and reports other nodes working on the network.
- Destination Node: It performs a scan in the network to look for a specific node.
- Duplicate Detection: 802.15.4 does not detect duplicate packets. Using Digi extra header, this problem is solved, discarding automatically the duplicate packets. If Digi header is not enabled, the application level will have to deal with this problem.

These features and extra header are enabled by default. Possible values for this parameter are:

- 0: Digi Mode. 802.15.4 header + Digi header. It enables features as Discover Node and Destination Node.
- 1: 802.15.4 without ACKs. It doesn't support DN and ND features. It is 802.15.4 protocol without generating ACKs when a packet is received.
- 2: 802.15.4 with ACKs. It doesn't support DN and ND features. It is the standard 802.15.4 protocol.
- 3: Digi Mode without ACKs. 802.15.4 header + Digi header. It enables features as Discover Node and Destination Node. It doesn't generate ACKs when a packet is received.

Example of use:

```
{
  xbee802.setMacMode(2); // Set Mac Mode to 802.15.4 header
  xbee802.getMacMode(); // Get Mac Mode
}
```

Related Variables

xbee802.macMode ⟶ stores the Mac Mode selected

# 6. Power Gain and Sensibility

When configuring a node and a network, one important parameter is related with power gain and sensibility.

## 6.1. Power Level

Power level(dBm) at which the RF module transmits conducted power. Its possible values are:

| Parameter | XBee-PRO | XBee-PRO International |
|---|---|---|
| 0 | 10dBm | PL=4 : 10dBm |
| 1 | 12dBm | PL=3 : 8dBm |
| 2 | 14dBm | PL=2 : 2dBm |
| 3 | 16dBm | PL=1 : -3dBm |
| 4 | 18dBm | PL=0 : -3dBm |

*Figure 10: Power Output Level*

***Note:*** *dBm is a standard unit to measure power level taking as reference a 1mW signal.*
Values expressed in dBm can be easily converted to mW using the next formula :

mW = 10^(value dBm/10)

Graphic about transmission power is exposed next:

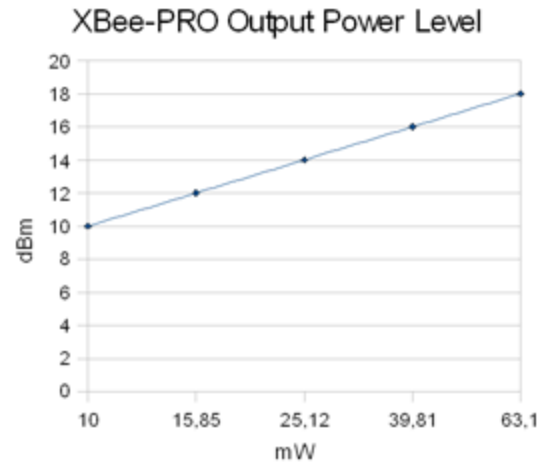



*Figure 11: XBee-PRO Output Power Level*

Example of use

```
{
  xbee802.setPowerLevel(0); // Set Power Output Level to the minimum value
  xbee802.getPowerLevel(); // Get Power Output Level
}
```

Related Variables

xbee802.powerLevel ⟶ stores the power output level selected

Power Level configuration example:

**http://www.libelium.com/development/waspmote/examples/802-15-setread-power-level**

# 6.2. Received Signal Strength Indicator

It reports the Received Signal Strength of the last received RF data packet. It only indicates the signal strength of the last hop, so it does not provide an accurate quality measurement of a multihop link.

Example of use:

```
{
  xbee802.getRSSI(); // Get the Receive Signal Strength Indicator
}
```

Related Variables

xbee802.valueRSSI ⟶ stores the RSSI of the last received packet

The returned command value is measured in -dBm. For example if xbee802.valueRSSI=0x60, then the RSSI of the last packet received was -96dBm. The ideal working mode is getting the maximum coverage with the minimum power level. Thereby, a compromise between power level and coverage appears. Each application scenario will need some tests to find the best combination of both parameters.

• Getting RSSI example:

**http://www.libelium.com/development/waspmote/examples/802-08-get-rssi**

# 7. Radio Channels

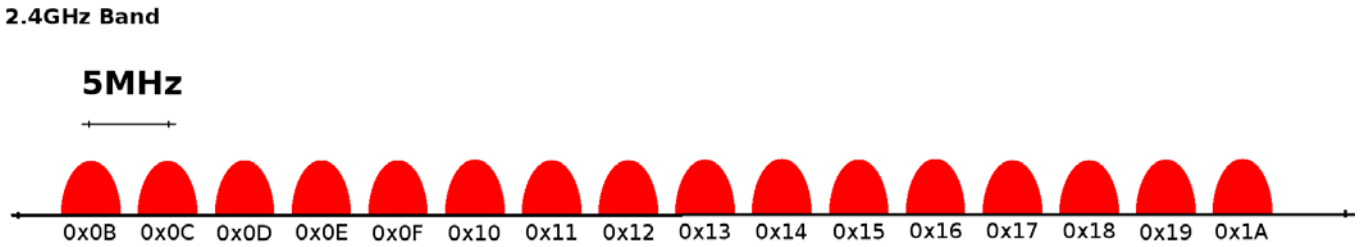



*Figure 12: Frequency Channels on 2.4GHz Band*

The XBee 802.15.4 module works on 2.4GHz band, so there are 12 channels available. The values are between 0x0C-0x17(12values).

Due to the frequency band used is a free band, a channel may be jammed by other network transmissions. To avoid that jamming, it is recommended to scan all the channels to choose the channel with less energy.

## 7.1. Scan Channels

List of channels to scan for all Active and Energy Scans as a bitfield:

| | | | |
|---|---|---|---|
| bit 0: 0x0B | bit 4: 0x0F | bit 8: 0x13 | bit 12: 0x17 |
| bit 1: 0x0C | bit 5: 0x10 | bit 9: 0x14 | bit 13: 0x18 |
| bit 2: 0x0D | bit 6: 0x11 | bit 10: 0x15 | bit 14: 0x19 |
| bit 3: 0x0E | bit 7: 0x12 | bit 11: 0x16 | bit 15: 0x1A |

Setting this parameter to 0xFFFF, will cause all the channels to be scanned.

The API functions store the list in a related array called 'scanChannels'.

Example of use

```
{
  xbee802.setScanningChannels(0xFF,0xFF); // Sets all channels to scan
  xbee802.getScanningChannels(); // Gets scanned channel list
}
```

Related Variables

xbee802.scanChannels[0-1] ⟶ stores the list of channels to scan

## 7.2. Energy Scan

The maximum energy on each channel is returned, starting with the first channel selected in Scan Channels parameter. The amount of time this energy scan is performed is specified as an API function input. This time is calculated in this way (2^ED)*15,36ms.

The recommended process for choosing a free channel is:

1. Change the list of channels to scan all the channels.
2. Perform an energy scan channel on those channels.
3. A channel is chosen among the scanned channels, selecting the channel with minimum energy.

Example of use:

```
{
  xbee802.setDurationEnergyChannels(3); // Performs the energy scan
}
```

Related Variables

xbee802.energyChannel[0-15] ⟶ stores the energy found on each scanned channel

• Energy scan example:

**http://www.libelium.com/development/waspmote/examples/802-07-energy-scan**

***Note**: The amount of time specified to scan each channel may affect the detected energy. In our tests, selecting the minimum value was not sufficient to detect the energy, so it is recommended to select a higher value.*

***Note 2**: After testing, a usual energy value for a free channel is around -84dBm and -37dBm for a occupied one. When the energy scan is performed, these values may be used to determine wether a channel is occupied or not.*

This protocol provides some parameters to diagnose the current state of the network. 802.15.4 uses **CSMA-CA** to avoid various nodes start transmitting at the same time. The process is described next:

1. A random delay is waited for. This time is specified by a back-off algorithm.
2. Performs a Clear Channel Assessment.
3. If the detected energy is above the CCA threshold, steps 1-3 will be repeated up to 4 times. If the detected energy is under the CCA threshold, the packet is transmitted.
4. Broadcast transmissions has already finished because ACKs are not sent. Unicast transmissions wait for ACK. If ACK is not received, steps 1-4 will be repeated up to 3 times.

## 7.3. Random Delay

It specifies the minimum value at which the back-off algorithm starts. First time this algorithm is executed, it waits a random delay: (2^BE-1)*0.32ms. BE is the back-off algorithm exponent and it starts at the value 'Random Delay'. In the same attempting of sending a packet, the exponent is increased in one.

Example of use:

```
{
  xbee802.setDelaySlots(1); // Sets the exponent to start
  xbee802.getDelaySlots(); // Gets the exponent to start
}
```

Related Variables

xbee802.delaySlots ⟶ stores the exponent

## 7.4. CCA Threshold

Clear Channel Assessment Threshold. Prior to transmitting a packet, a CCA is performed to detect energy on the channel. If the detected energy is above the CCA Threshold, the module will not transmit the packet. By default, this value is -44dBm. In our tests, an occupied channel with some nodes transmitting has around -37dBm, so it is a tighten value.

Example of use:

```
{
  xbee802.setEnergyThreshold(0x2C); // Sets the threshold
  xbee802.getEnergyThreshold (); // Gets the threshold
}
```

Related Variables

xbee802.energyThreshold ⟶ stores the energy threshold

## 7.5. CCA Failures

Counter of Clear Channel Assessment in CSMA-CA process. It specifies the number of times a packet has been discarded due to the energy on the channel was above the CCA Threshold.

Example of use:

```
{
  xbee802.getCCAcounter(); // Gets the CCA counter
  xbee802.resetCCAcounter(); // Resets the CCA counter
}
```

Related Variables

xbee802.counterCCA[0-1] ⟶ stores the CCA counter

## 7.6. ACK Failures

Counter of acknowledgment failures. It specifies the number of times a packet has been sent and its ACK has not been received.

Example of use:

```
{
  xbee802.getACKcounter(); // Get the ACK counter
  xbee802.resetACKcounter(); // Reset the ACK counter
}
```

Related Variables

xbee802.counterACK[0-1] ⟶ stores the ACK counter

## 7.7. Retries

Maximum number of retries the module will execute **in addition** to the 3 retries specified by 802.15.4 protocol. For each XBee retry, the 802.15.4 can execute up to 3 retries. If the transmitting module does not receive an ACK after 200ms, it will re-send the packet up to 3 times. This is an upper-MAC level, so the CSMA-CA process explained before is executed each retry.

Example of use:

```
{
  xbee802.setRetries(2); // Set the additional retries
  xbee802.getRetries(); // Get the additional retries
}
```

Related Variables

xbee802.retries[0-1] ⟶ stores the retries

Using these parameters, a bad behaving of a channel could be detected, and the process to choose a new channel should be executed.

# 8. Connectivity

## 8.1. Topologies

802.15.4 provides a star topology to create a network:

- **Star:** a star network has a central node, which is linked to all other nodes in the network. The central node gathers all data coming from the network nodes.

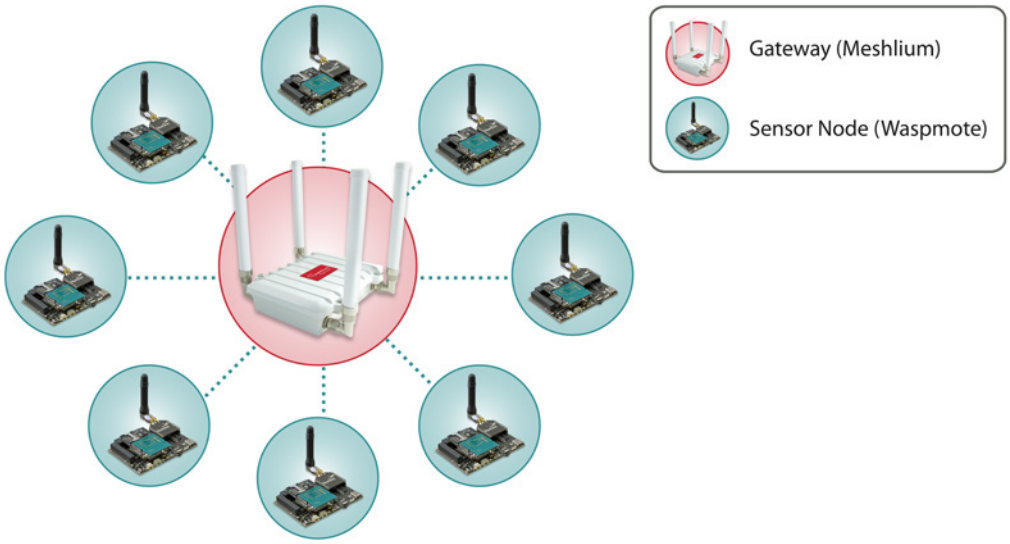



*Figure 13: Star Topology*

## 8.2. Connections

Every RF data packet sent over-the-air contains a Source Address and Destination Address field in its header. The RF module conforms to the 802.15.4 specification and supports both short 16-bit addresses and long 64-bit addresses. A unique 64-bit IEEE source address is assigned at the factory and can be read with the functions explained in chapter 4. Short addressing must be configured manually.

IEEE 802.15.4 supports unicast and broadcast transmission.

### 8.2.1. Unicast

Unicast Mode is the only one that supports retries. While in this mode, receiving modules send an ACK of RF packet reception to the transmitter. If the transmitting module does not receive the ACK, it will re-send the packet up to three times or until the ACK is received.

16-bit addresses and 64-bit addresses are supported in unicast mode. To use 16-bit addresses, the transmitter should set the receiver network address into the transmitted packet. To use 64-bit addresses, the transmitter should set the receiver MAC address into the transmitted packet and the receiver network address must be set to '0xFFFF'.

### 8.2.2. Broadcast

Used to send a packet to all the nodes in a network. Any RF module within range will accept a packet that contains a broadcast address. When configured to operate in the Broadcast Mode, receiving modules do not send ACKs and transmitting modules do not automatically re-send packets as the case in Unicast Mode. To send a broadcast message, the 64-bit destination address should be set to 0x000000000000FFFF.

## 8.3. Sending Data

Sending data is a complex process which needs some special structures and functions to carry out. Due to the limit on maximum payloads (see chapter "Packet Parameters") it could happen that the packet has to be truncated to the maximum possible length.

### 8.3.1. XBee API Frame Structure

API mode (AP=2) is set in XBee modules in order to work with the XBee API frame structure. The API frame structure used to transmit packets via RF with XBee modules is specified in Digi's product manual. Depending on using 64-bit or 16-bit transmissions, API Frame structure will be different. The following figure shows how the payload is included inside the RF Data field of the XBee API frame structure:

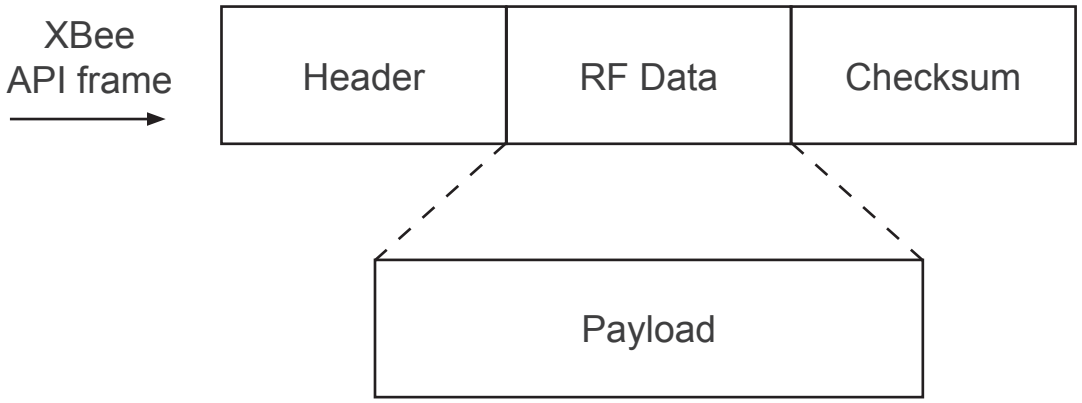



*Figure 14: XBee API Frame Structure*

The frame's header includes information about the destination address, transmission mode, etc. which are set by Waspmote's libraries when a sending is done. The sending process is defined in the following points. This header is not explained in detail inside this guide. For more information please check the manufacturer's manual.

### 8.3.2. Using WaspFrame class to create XBee frames

WaspFrame is a class that allows the user to create data frames with a specified format. It is a very useful tool to set the payload of the packet to be sent. It is recommended to read the Waspmote Frame Programming Guide in order to understand the XBee examples:

**http://www.libelium.com/development/waspmote/documentation/programming**

### 8.3.3. Sending

This is the process to send a packet between Waspmote devices.

**Create a new packet**

A structure 'packetXBee' is created to contain the packet to send:

```
packetXBee* packet;
packet=(packetXBee*) calloc(1,sizeof(packetXBee));
```

**Select the transmission mode**

Transmission mode. Select between:

UNICAST: `packet->mode=UNICAST;`

BROADCAST: `packet->mode=BROADCAST;`

**Fill destination parameters**

'setDestinationParams' function sets the destination parameters to the packet which is going to be sent. The following parameters are set:

- Destination address. It depends on the type of destination address. There are two possibilities:
  - MAC_TYPE: It sets the macDL and macDH packet parameters. MAC address can be specified manually calling the function this way:

```
char* mac_address="0013A2004030F6BC";
xbee802.setDestinationParams(packet, mac_address, data, MAC_TYPE);
```

  - MY_TYPE: It sets the naD packet parameter, also called MY address. It can be specified manually calling the function this way:

```
char* network_address="1234";
xbee802.setDestinationParams(packet, network_address, data, MY_TYPE);
```

- Data field to be sent within the packet. Data field can be defined as three different possibilities:
  - Array of characters:

```
char* data="This is the data field";
xbee802.setDestinationParams(packet, MAC, data, MAC_TYPE);
```

  - Array of bytes (it is mandatory to specify the length of the data field):

```
uint8_t data[5]={0x00, 0x01, 0x54, 0x76, 0x23};
xbee802.setDestinationParams(packet, MAC, data, 5, MAC_TYPE);
```

  - Integer value (integer value is converted to an array of characters):

```
int data=245;
xbee802.setDestinationParams(packet, MAC, data, MAC_TYPE);
```

**Send data**

The API function responsible of sending data is called, using the previously created structure as input:

```
xbee802.sendXBee(packet);
```

• Sending example:

**http://www.libelium.com/development/waspmote/examples/802-02-send-packets**

## 8.3.4. Examples

• Send packets in unicast mode. Set 16-bit destination address (MY_TYPE):

**http://www.libelium.com/development/waspmote/examples/802-04a-send-unicast-16b-address**

• Send packets in unicast mode. Set 64-bit destination address (MAC_TYPE):

**http://www.libelium.com/development/waspmote/examples/802-05a-send-unicast-64b-address**

• Send packets in broadcast mode:

**http://www.libelium.com/development/waspmote/examples/802-06a-send-broadcast**

• Send packets using the expansion board:

**http://www.libelium.com/development/waspmote/examples/802-09a-expansion-board-send**

• Complete example, send packets in unicast 64-bit addresing mode and wait for a response:

**http://www.libelium.com/development/waspmote/examples/802-11a-complete-example-send**

# 8.4. Receiving Data

Receiving data is a complex process which needs some special structures to carry out. These operations are transparent to the API user, so it is going to be explained the necessary information to be able to read properly a received packet.

Before any packet has been received an array of 'packetXBee' structures pointers are created. This array is called packet_finished. The size of this array is defined by a constant in WaspXBeeCore.h file in compilation time, so it is necessary to adequate its value to each scenario before compile the code:

- MAX_FINISH_PACKETS: (default value is 5) max number of finished packets pending of treatment. It specifies the size of the array of finished packets.

## 8.4.1. Waspmote API receiving procedure

When a packet is received and the proper API function is called, a 'packetXBee' structure is created and linked to the corresponding position in the array of finished packets structure. All information is extracted from the received frame from the XBee module and copied to this API structure fields. Thus, the packet data is available for the user.

## 8.4.2. How to receive packets in Waspmote

1. When a packet is received via RF, the module will send the data via UART, so it is recommended to check periodically if data is available. The API function responsible for reading packets can read more than one in a time, but the XBee module may overflow its buffer, so it is recommended to read packets one by one.

2. If there is available data, it must be treated in order to parse the information in packet structures.

3. If reception is correct, 'error_RX' flag will be set to '0' and the packet will be stored in a finished packets array called 'packet_finished'. This array should be used by the application to read the received packet.

4. A variable called 'pos' is used to know if there are received packets to be processed by the user: if 'pos'=0, there is no available packet; if 'pos'>0, there will be as many pending packets as its value (pos=3 means 3 pending packets).

5. For each 'pos' value, a packet is available in 'xbee802.packet_finished' structure. So it is possible to access to all fields which have been seen in chapter "Packet Parameters".

6. Once a packet is used as the user wants, it is necessary to free the allocated memory to this packet and decrement the number of available packets indicated by 'pos'.

• Receiving packets example:

**http://www.libelium.com/development/waspmote/examples/802-03-receive-packets**

***Note**: If no API header mode is used to transmit, the receiving API features are not available. Remember that mode has only been designed to transmit directly to a gateway.*

### 8.4.3. Examples

• Receive packets in unicast mode. Get 16-bit source address (MY_TYPE):

**http://www.libelium.com/development/waspmote/examples/802-04b-receive-unicast-16b-address**

• Receive packets in unicast mode. Get 64-bit source address (MAC_TYPE):

**http://www.libelium.com/development/waspmote/examples/802-05b-receive-unicast-64b-address**

• Receive packets in broadcast mode (the same procedure as if it was unicast mode):

**http://www.libelium.com/development/waspmote/examples/802-06b-receive-broadcast**

• Receive packets using the expansion board:

**http://www.libelium.com/development/waspmote/examples/802-09b-expansion-board-reception**

• Complete example, receive packets and send a response back to the sender:

**http://www.libelium.com/development/waspmote/examples/802-11b-complete-example-receive**

# 9. Starting a Network

To create a network only two parameters are necessary: **channel** and **PAN ID**. These parameters are the base of a network and we need to be careful choosing them. There are more parameters used to create a network like security parameters, which are not necessary but recommended (see chapter 12).

Two nodes are in the same network if they are using the same channel and PAN ID.

## 9.1. Choosing a channel

As explained in chapter "Node Parameters", there are different channels to choose. A random value between '0x0C'-'0x17' should be chosen. This value will be used as the input parameter in the API function responsible of setting the channel.

Example of use

```
{
   xbee802.setChannel(0x0C); // Set channel
}
```

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

## 9.2. Choosing a PAN ID

A network must have a unique PAN ID. This PAN ID is a 16-bit number whose values are comprised between 0-0xFFFF.

To set a valid PAN ID it is necessary to select a random number and use the API function responsible of setting this parameter. Any value comprised between 0-0xFFFF could be valid, but it should be used only by one network to avoid conflict problems.

Example of use

```
{
  uint8_t PANID[2]={0x33,0x31}; // array containing the PAN ID
  xbee802.setPAN(PANID); // Set PANID
}
```

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

***Note**: If two different networks are using the same PAN ID, it doesn't mean there will be interferences between them. Interferences will appear if both networks are transmitting in the same channel. If that happens, it will mean the two different networks will become the same network because both are using same PAN ID and channel frequency.*

# 10. Joining an Existing Network

Joining an existing network process requires some knowledge about the network to join. Three parameters are needed: channel, PAN ID and security (explained in chapter "Security and Data Encryption").

## 10.1. Channel

To set channel, use the API function responsible of that matter.

Example of use

```
{
  xbee802.setChannel(0x10); // Set Channel the network is working on
}
```

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

## 10.2. PAN ID

To set PAN ID, use the API function responsible of that matter.

Example of use

```
{
  uint8_t PANID[2]={0x33,0x31}; // array containing PAN ID the network is using
  xbee802.setPAN(PANID); // Set PANID
}
```

Once these two parameters are set in the node, it will be part of the network, receiving messages from other nodes.

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

## 10.3. Node Discovery

XBee modules provide some features for discovering and searching nodes. These features added to 802.15.4 allow a node to send a broadcast message to discover other nodes in the network within its coverage.

### 10.3.1. Structure used in Node Discovery

Discovering nodes is used to discover and report all modules on its current operating channel and PAN ID.

To store the reported information by other nodes, an structure called 'Node' has been created. This structure has the next fields:

```
uint8_t MY[2];    // 16b Network Address
uint8_t SH[4];    // 32b Lower Mac Source
uint8_t SL[4];    // 32b Higher Mac Source
char NI[20];      // Node Identifier
uint8_t PMY[2];   // Parent 16b Network Address
uint8_t DT;       // Device Type: 0=End 1=Router 2=Coordinator
uint8_t ST;       // Status: Reserved
uint8_t PID[2];   // Profile ID
uint8_t MID[2];   // Manufacturer ID
uint8_t RSSI;     // Receive Signal Strength Indicator
```

**• MY**

16-bit Network Address of the reported module.

**• SH & SL**

64-bit MAC Source Address of the reported module.

**• NI**

Node Identifier of the reported module

**• PMY & DT & ST & PID & MID**

Not used in 802.15.4.

**• RSSI**

Received Signal Strength Indicator of the received packet. This parameter is quite a lot important since it indicates the link quality between the two nodes.

To store the found brothers, an array called 'scannedBrothers' has been created. It is an array of structures 'Node'. To specify the maximum number of found brothers, it is defined a constant called 'MAX_BROTHERS'. It is also defined a variable called 'totalScannedBrothers' that indicates the number of brothers have been discovered. Using this variable as index in the 'scannedBrothers' array, it will be possible to read the information about each node discovered.

Example of use:

```
{
     xbee802.scanNetwork(); // Discovery nodes
}
```

Related variables

xbee802.totalScannedBrothers → stores the number of brothers have been discovered.

xbee802.scannedBrothers → 'Node' structure array that stores in each position the info related to each found node. For example, xbee802.scannedBrothers[0].MY should store the 16-bit Network Address of the first found node.

• Scan network example:

**http://www.libelium.com/development/waspmote/examples/802-13-scan-network**

## 10.3.2. Searching specific nodes

Another possibility is searching for a specific node. This search is based on using the Node Identifier. The NI of the node to discover is used as the input in the API function responsible of this purpose. This function provides the 16-bit network address of the searched node.

Example of use

```
{
  uint8_t naD[2];
  xbee802.nodeSearch("forestNode-01", naD);
}
```

Related variables

naD[0] and naD[1] → Store the 16-bit address of the searched node

• Node search example:

**http://www.libelium.com/development/waspmote/examples/802-14-node-search**

### 10.3.3. Node discovery to a specific node

When executing a Node Discovery all the nodes respond to it. If its Node Identifier is known, a Node Discovery using its NI as an input can be executed.

Example of use

```
{
  xbee802.scanNetwork("forestNode-01"); // Performs a ND to that specific node
}
```

Related variables

xbee802.totalScannedBrothers → stores the number of brothers which have been discovered. In this case its value will be '1'

xbee802.scannedBrothers → 'Node' structure array that stores in each position the info related to each found node. It will store in the first position of the array the information related to the found brother

### 10.3.4. Node Discovery Time

It is the amount of time (NT) a node will wait for responses from other nodes when performing a ND. Range: 0x01 - 0xFC [x 100 ms]. Default: 0x19.

Example of use:

```
{
  uint8_t time[2]={0x19,0x00};// In 802.15.4 is only used first array position
  xbee802.setScanningTime(time); // Set Scanning Time in ND
  xbee802.getScanningTime(); // Get Scanning Time in ND
}
```

Available Information

xbee802.scanTime[0]→ stores the time a node will wait for responses.

### 10.3.5. Node Discover Options

Enables node discover self-response on the module.

Example of use:

```
{
  xbee802.setDiscoveryOptions(); // Set Discovery Options for ND
  xbee802.getDiscoveryOptions(); // Get Discovery Options for ND
}
```

Available Information

xbee802.discoveryOptions → stores the selected options for ND

***Note**: If Node Discovery Time is a long time, the API function may exit before receiving a response, but the module will wait for it, crashing the code. If this time is too short, it is possible the modules don't answer. After testing several times, the best values are between 1-2 seconds, setting the API function appropriately.*

## 10.3.6. Questions related to Discovering Nodes

While testing, many questions came up about discovering nodes. Now these questions are going to be exposed an explained to help the developer.

**• What are the network parameters needed to perform a ND?**

After testing a network, PAN ID and channel are needed to perform a ND. If one of these parameters is unknown, when performing a ND no node will response. If security is enabled in a network, the security key will be needed too.

**• Is a node obliged to answer a ND? DoS Attack.**

Yes, it is obliged to answer. The only way to prevent a network form DoS attack is using encryption, since the attacker will have to know the key to perform the ND.

**• What happens when ND is performed using PANID=0xFFFF?**

When performing a ND using this PANID no response is received. Broadcast PANID is not as useful as in a broadcast message, since the nodes does not receive any message.

**• Node Discover Time. Tests.**

By default, this parameter is 0x19(2,5seconds). This is a great value, that can be reduced to make the ND process shorter. Tests have demonstrated this value can be reduced up to 0,5 seconds and the nodes still answer.

However, it is recommended not to reduce this parameter so much. When reducing this time, a node may answer after this time, the code crashing.

During this discovery time, the module is waiting for an answer and will not accept any data via serial port, so the API function 'scanNetwork' will have to be modified if this parameter is changed. If it is not modified, the API function will exit but the module will be still waiting, making the next functions don't work.

If this discovery time is reduced and a node answers when the module and the API function have exit ND, the next called function will not work because the data sent by the module via serial will correspond with the response of ND.

# 11. Sleep Options

Sleep Modes enable the RF module to enter into states of low-power consumption when not in use. To set sleep mode on, there are some parameters involved.

## 11.1. Sleep Mode

By default, Sleep Modes are disabled and the RF module remains in Idle/Receive Mode. When in this state, the module is constantly ready to respond to either serial or RF activity.

Different options can be set:

- 0: Disabled.
- 1: Pin Hibernate Mode.
- 2: Pin Doze Mode.

Example of use:

```
{
  xbee802.setSleepMode(0); // Disable sleep mode
  xbee802.setSleepMode(1); //  Set Sleep Mode to Pin Sleep
  xbee802.getSleepMode(); // Get the Sleep Mode used
}
```

Related Variables

xbee802.sleepMode ⟶ stores the sleep mode in a module

• Sleep mode example:

**http://www.libelium.com/development/waspmote/examples/802-10-set-low-power-mode**

## 11.2. Pin Hibernate Mode

Pin Hibernate Mode minimizes power consumption (<10uA). This mode is voltage level-activated; when Sleep_RQ (pin 9 of XBee) is asserted, the module will finish any transmission, reception or association activities, enter Idle Mode, and then enter a state of sleep. The module will not respond to either serial or RF activity while in pin sleep.

To wake up an sleeping module operating in Pin Hibernate Mode, de-assert Sleep_RQ (pin 9 of XBee). The module will be awake after 13.2ms.

Example of use:

```
{
  xbee802.setSleepMode(1); // Set Sleep Mode to Hibernate Mode
  xbee802.sleep(); // Set XBee to sleep

  delay(5000); // wait 5 seconds
  xbee802.wake(); // Wake up the XBee
}
```

## 11.3. Pin Doze Mode

Pin Doze Mode works as Pin Hibernate Mode. However, Pin Doze features faster wake-up time (2ms) and higher power consumption (<50uA).

Example of use:

```
{
  xbee802.setSleepMode(2); // Set Sleep Mode to Doze Mode
  xbee802.sleep(); // Set XBee to sleep
  delay(5000); // wait 5 seconds
  xbee802.wake(); // Wake up the XBee
}
```

## 11.4. ON/OFF Modes

In addition to the XBee sleep modes, Waspmote provides the feature of controlling the power state with a digital switch. This means that using one function included in Waspmote API, any XBee module can be powered up or down (0uA). This function does not open/close the UART related to XBee module.

Example of use:

```
{
   xbee802.setMode(XBEE_ON); // Powers XBee up
   xbee802.setMode(XBEE_OFF); // Powers XBee down
}
```

# 12. Security and Data Encryption

## 12.1. IEEE 802.15.4 Security and Data encryption Overview

The encryption algorithm used in 802.15.4 is AES (Advanced Encryption Standard) with a 128b key length (16 Bytes). The AES algorithm is not only used to encrypt the information but to validate the data which is sent. This concept is called **Data Integrity** and it is achieved using a Message Integrity Code (MIC) also named as Message Authentication Code (MAC) which is appended to the message. This code ensures integrity of the MAC header and payload data attached.

It is created encrypting parts of the IEEE MAC frame using the Key of the network, so if we receive a message from a non trusted node we will see that the MAC generated for the sent message does not correspond to the one what would be generated using the message with the current secret Key, so we can discard this message. The MAC can have different sizes: 32, 64, 128 bits, however it is always created using the 128b AES algorithm. Its size is just the bits length which is attached to each frame. The more large the more secure (although less payload the message can take). **Data Security** is performed encrypting the data payload field with the 128b Key.

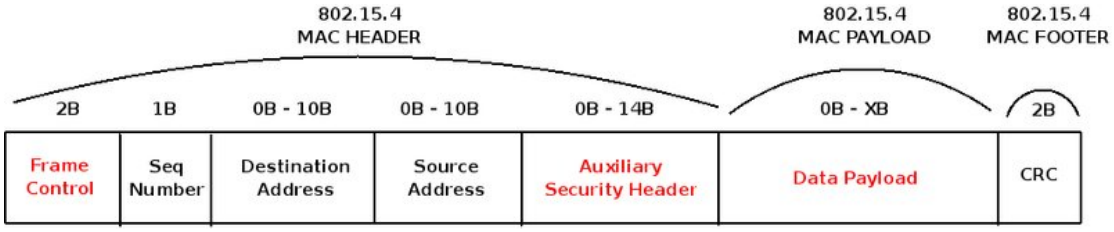



*Figure 15: IEEE 802.15.4 Frame*

## 12.2. Security in API libraries

As explained previously, IEEE 802.15.4 provides secure communications inside a network using 128-bit AES encryption. The API functions enable using security and data encryption.

### 12.2.1. Encryption Enable

Enables the 128-bit AES encryption in the modules.

When encryption is enabled, the module will always use its 64-bit address as the source address for RF packets. With encryption enabled and a 16-bit address set, receiving modules will only be able to issue receive 64-bit indicators.

Example of use:

```
{
  xbee802.setEncryptionMode(0); // Disable encryption mode
  xbee802.setEncryptionMode(1); // Enable encryption mode
  xbee802.getEncryptionMode(); // Get encryption mode
}
```

Related Variables

xbee802.encryptMode ⟶ stores if security is enabled or not

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

The mode used to encrypt the information is AES-CTR. In this mode all the data is encrypted using the defined 128b key and the AES algorithm. The Frame Counter sets the unique message ID and the Key Counter (Key Control subfield) is used by the application layer if the Frame Counter max value is reached.

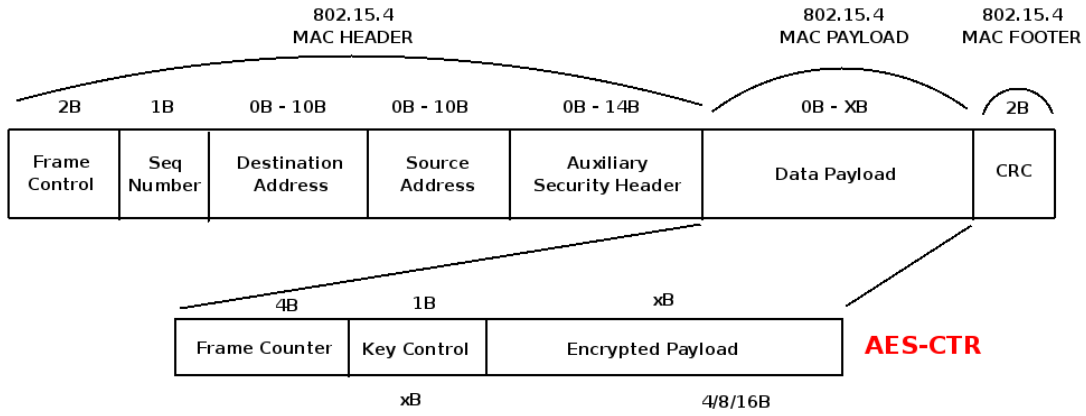



*Figure 16: AES-CTR Encryption Frame*

### 12.2.2. Encryption Key

128-bit AES encryption key used to encrypt/decrypt data.

The entire payload of the packet is encrypted using the key and the CRC is computed across the ciphertext. When encryption is enabled, each packet carries an additional 16 Bytes to convey the random CBC Initialization Vector (IV) to the receivers.

A module with the wrong key (or no key) will receive encrypted data, but the data driven out the serial port will be meaningless. A module with a key and encryption enabled will receive data sent from a module without a key and the correct unencrypted data output will be sent out the serial port.

Example of use

```
{
  char* KEY="WaspmoteLinkKey!"
  xbee802.setLinkKey(KEY); // Set Encryption Key
}
```

Related Variables

xbee802.linkKey ⟶ stores the key that has been set in the network

• XBee configuration example:

**http://www.libelium.com/development/waspmote/examples/802-01-configure-xbee-parameters**

## 12.3. Security in a network

When creating or joining a network, using security is highly recommended to prevent the network from attacks or intruder nodes.

It is necessary to enable security and set the same encryption key in all nodes in order to set security in a network. If not, it won't be possible to communicate between different XBee modules.

# 13. Code examples and extended information

In the Waspmote Development section you can find complete examples:

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

Example:

```
#include <WaspXBee802.h>
#include <WaspFrame.h>

//Pointer to an XBee packet structure
packetXBee* packet;

// Destination MAC address
char* MAC_ADDRESS="0013A2004052414C";

void setup()
{
  // init USB port
  USB.ON();
  USB.println(F("802_2 example"));

  // init XBee
  xbee802.ON();

  // set mote Identifier (16-Byte max)
  frame.setID("WASPMOTE_XBEE");
}

void loop()
{
  // create new frame
  frame.createFrame();

  // add frame fields
  frame.addSensor(SENSOR_STR, "XBee frame");
  frame.addSensor(SENSOR_BAT, PWR.getBatteryLevel());

  // set parameters to packet:
  packet=(packetXBee*) calloc(1,sizeof(packetXBee)); // Memory allocation
  packet->mode=UNICAST; // Choose transmission mode: UNICAST or BROADCAST

  // set destination XBee parameters to packet
  xbee802.setDestinationParams( packet, MAC_ADDRESS, frame.buffer, frame.length, MAC_TYPE);

  // send XBee packet
  xbee802.sendXBee(packet);

  // check TX flag
  if( xbee802.error_TX == 0 )
  {
    USB.println(F("ok"));
  }
  else
  {
    USB.println(F("error"));
  }

  // free variables
  free(packet);
  packet=NULL;

  // wait for five seconds
  delay(5000);
}
```

# 14. API changelog

| Function / File | Changelog | Version |
|---|---|---|
| WaspXBeeCore::getPowerLevel | New function | v005 → v006 |
| WaspXBeeCore::getRSSI | Internal change in function | v005 → v006 |
| WaspXBeeCore::getHardVersion | Internal change in function | v005 → v006 |
| WaspXBeeCore::getSoftVersion | Internal change in function | v005 → v006 |
| WaspXBeeCore::setRSSItime | Internal change in function | v005 → v006 |
| WaspXBeeCore::getRSSItime | Internal change in function | v005 → v006 |
| WaspXBeeCore::applyChanges | Internal change in function | v005 → v006 |
| WaspXBeeCore::reset | Internal change in function | v005 → v006 |
| WaspXBeeCore::resetDefaults | Internal change in function | v005 → v006 |
| WaspXBeeCore::setSleepOptions | Internal change in function | v005 → v006 |
| WaspXBeeCore::getSleepOptions | Internal change in function | v005 → v006 |
| WaspXBeeCore::scanNetwork | Internal change in function | v005 → v006 |
| WaspXBeeCore::setDurationEnergyChannels | Internal change in function | v005 → v006 |
| WaspXBeeCore::getDestinationAddress | Internal change in function | v005 → v006 |
| WaspXBeeCore::new_firmware_received | Internal change in function | v005 → v006 |
| WaspXBeeCore::new_firmware_packets | Internal change in function | v005 → v006 |
| WaspXBeeCore::new_firmware_end | Internal change in function | v005 → v006 |
| WaspXBeeCore::upload_firmware | Internal change in function | v005 → v006 |
| WaspXBeeCore::request_bootlist | Internal change in function | v005 → v006 |
| WaspXBeeCore::checkNewProgram | Internal change in function | v005 → v006 |
| WaspXBeeCore::delete_firmware | Internal change in function | v005 → v006 |
| WaspXBeeCore::checkOtapTimeout | Internal change in function | v005 → v006 |
| WaspXBee class | This class no longer exists | v0.31 → v001 |
| WaspXBeeCore::begin(<br>uint8_t uart,<br>uint32_t speed); | New function moved from WaspXBee class to WaspXBeeCore class. | v0.31 → v001 |
| WaspXBeeCore::setMode(uint8_t mode); | New function moved from WaspXBee class to WaspXBeeCore class. | v0.31 → v001 |
| WaspXBeeCore::close(); | New function moved from WaspXBee class to WaspXBeeCore class. | v0.31 → v001 |
| WaspXBeeConstants.h | WaspXBeeConstants.h file is deleted. All constants are now defined in Flash memory so as to save RAM memory. | v0.31 → v001 |
| WaspXBeeCore::init(<br>uint8_t protocol_used,<br>uint8_t frequency,<br>uint8_t model_used,<br>uint8_t uart_used); | This function is no longer defined inside WaspXBeeCore.cpp. It is now declared as virtual function in order to be re-defined in derived classes. Then, this function is not longer used in Waspmote codes when initializing the XBee modules. Now, it is only necessary to call the correspondent object's ON function. | v0.31 → v001 |

| | | |
|---|---|---|
| `WaspXBeeCore::init(`<br>`uint8_t protocol_used,`<br>`uint8_t frequency,`<br>`uint8_t model_used);` | This function is no longer defined inside WaspXBeeCore.cpp. It is now declared as virtual function in order to be re-defined in derived classes. Then, this function is not longer used in Waspmote codes when initializing the XBee modules. Now, it is only necessary to call the correspondent object's ON function. | v0.31 → v001 |
| `WaspXBeeCore::ON` | This function has been changed in order to support SOCKET selection. It is possible to select between both sockets, i.e. xbee802.ON(SOCKET0); or xbee802.ON(SOCKET1); | v0.31 → v001 |
| `WaspXBeeCore::setDestinationParams` | New function prototype. For each setDestinationParams prototype there is a new variant which selects MAC_TYPE addressing as the default type. Also, the last parameter 'off_type' has been deleted in all prototypes. Now, each setDestinationParams call sets a new packet. There is no possibility of setting any data offset in the payload | v0.31 → v001 |
| `WaspXBeeCore::setDestinationParams(`<br>`packetXBee* paq,`<br>`uint8_t* address,`<br>`uint8_t* data,`<br>`int length,`<br>`uint8_t type);` | New function. When including the data field as byte array instead of strings, a new length parameter permits to specify the length of that byte array. | v0.31 → v001 |
| `WaspXBeeCore::setDestinationParams(`<br>`packetXBee* paq,`<br>`const char* address,`<br>`uint8_t* data,`<br>`int length,`<br>`uint8_t type );` | New function. When including the data field as byte array instead of strings, a new length parameter permits to specify the length of that byte array. | v0.31 → v001 |
| `uint8_t hops;` | 'hops' attribute has been deleted. This field will be set to 0x00. The NH parameter will permit the advanced users to set the number of hops (actually it is the timeout) in the network. | v0.31 → v001 |
| `#define DATA_MATRIX` | DATA_MATRIX definition has been deleted. | v0.31 → v001 |
| `#define TIMEOUT` | TIMEOUT definition has been deleted. | v0.31 → v001 |
| `#define NI_TYPE` | NI_TYPE definition has been deleted. | v0.31 → v001 |
| `#define DATA_ABSOLUTE` | DATA_ABSOLUTE definition has been deleted. | v0.31 → v001 |
| `#define DATA_OFFSET` | DATA_OFFSET definition has been deleted. | v0.31 → v001 |
| `uint16_t frag_length;` | 'frag_length' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `uint8_t macOL[4];` | 'macOL' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `uint8_t macOH[4];` | 'macOH' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `uint8_t naO[2];` | 'naO' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `char niO[21];` | 'niO' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `uint8_t typeSourceID;` | 'typeSourceID' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `uint8_t numFragment;` | 'numFragment' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `uint8_t endFragment;` | 'endFragment' variable from packetXBee struct has been deleted. | v0.31 → v001 |
| `int8_t WaspXBeeCore::send` | All send prototype functions have been deleted | v0.31 → v001 |

| | | |
|---|---|---|
| `uint8_t WaspXBeeCore::freeXBee` | 'freeXBee' function has been deleted | v0.31 → v001 |
| `uint8_t WaspXBeeCore::synchronization` | 'synchronization' function has been deleted | v0.31 → v001 |
| `int8_t WaspXBeeCore::setOriginParams` | All 'setOriginParams' prototype functions have been deleted | v0.31 → v001 |
| `uint16_t start;` | 'start' attribute has been deleted | v0.31 → v001 |
| `uint16_t finish;` | 'finish' attribute has been deleted | v0.31 → v001 |
| `uint16_t frag_length;` | 'frag_length' attribute has been deleted | v0.31 → v001 |
| `long TIME1;` | 'TIME1' attribute has been deleted | v0.31 → v001 |
| `WaspXBeeCore::readXBee(uint8_t* data);` | Function changed to virtual function so as to be re-defined in derived classes. | v0.31 → v001 |
| `WaspXBeeCore::SendXBeePriv( struct packetXBee* packet);` | Function changed to virtual function so as to be re-defined in derived classes. | v0.31 → v001 |
| `WaspXBeeCore::gen_frame` | protected API function renamed to WaspXBeeCore::genDataPayload. Now more fields are filled. | v0.31 → v001 |
| `WaspXBeeCore::genDataPayload` | New function in order to create all the RF Data payload in XBee frame. This payload is composed by the Waspmote API header (packetID + Fragment Number + First fragment Indicator (#) + Source Type ID + Source ID) and the Data field | v0.31 → v001 |
| `WaspXBeeCore::gen_escaped_frame( uint8_t* TX, uint8_t* data, int* final_length);` | New function in order to create escaped frames (with AP=2) for XBee AT command requests. | v0.31 → v001 |
| `WaspXBeeCore::txStatusResponse();` | Function prototype changed. And now this function is directly called inside the API when a transmission is done instead of including the calling inside parse_message function | v0.31 → v001 |
| `WaspXBeeCore::txZBStatusResponse( );` | Function prototype changed. And now this function is directly called inside the API when a transmission is doneinstead of including the calling inside parse_message function | v0.31 → v001 |
| `WaspXBeeCore::getChecksum(uint8_t* TX);` | New function in order to calculate the checksum of a frame before sending it to XBee module. Makes API easy to understand. | v0.31 → v001 |
| `uint8_t freq;` | Attribute doesn't exist any more | v0.31 → v001 |
| `uint8_t model;` | Attribute doesn't exist any more | v0.31 → v001 |
| `WaspXBeeCore::nodeSearch` | Function prototype has changed. Now a pointer to an array permits to store the destination address of the searched node. | v0.31 → v001 |
| `WaspXBeeCore::getRSSI` | Bug fixed. Added \r\n when sending the AT command. | v0.31 → v001 |
| `WaspXBeeCore::encryptionMode` | Function renamed to WaspXBeeCore::setEncryptionMode | v0.31 → v001 |
| `WaspXBeeCore::getEncryptionMode` | New Function in order to get the encryption mode | v0.31 → v001 |

# 15. Documentation changelog

**From v4.2 to v4.3**

- References to XBee 802.15.4 Normal / Standard version were deleted

**From v4.1 to v4.2**

- API changelog updated to API v006

**From v4.0 to v4.1**

- Added references to 3G/GPRS Board in section: Expansion Radio Board