Waspmote ZigBee

Networking Guide

libelium

waspmote

INDEX

1. Hardware

Module	Frequency	Transmission Power	Sensitivity	Number of channels	Distance
XBee-ZB-PRO	2,40 – 2,47GHz	50mW	-102dBm	14	7000m

Figure : XBee ZigBee PRO

Note: *From February 2014, Libelium no longer offers the XBee ZigBee wireless module in the Standard or Normal version. From that date, XBee ZigBee 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.*

As ZigBee is supported in the IEEE 802.15.5 link layer, it uses the same channels as described in the previous section, with the peculiarity that the XBee-ZB-PRO model limits the number of channels to 14.

The XBee-ZB modules comply with the **ZigBee-PRO v2007** standard. These modules add certain functionalities to those contributed by ZigBee, such as:

•	**Node discovery:** some headings are added so that other nodes within the same network can be discovered. It allows a node discovery message to be sent, so that the rest of the network nodes respond indicating their specific information (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.

The topologies in which these modules can be used are: star and tree.

Coordinator/ Gateway (Meshlium)

End Device/ Sensor Node (Waspmote)

Figure : Star topology

Coordinator/ Gateway (Meshlium)

Router (Waspmote)

End Device (Waspmote)
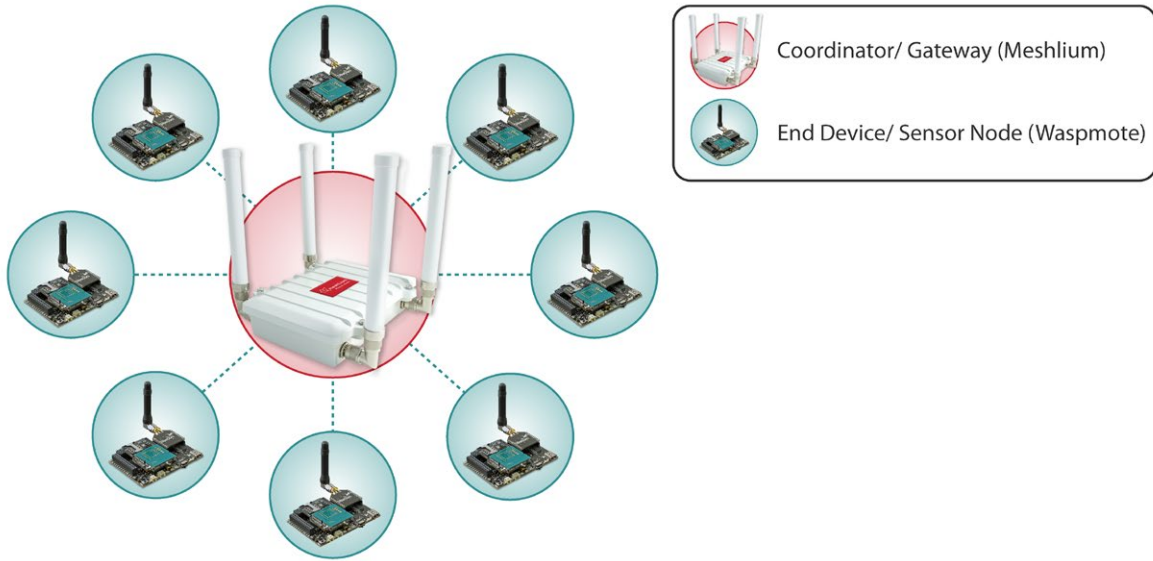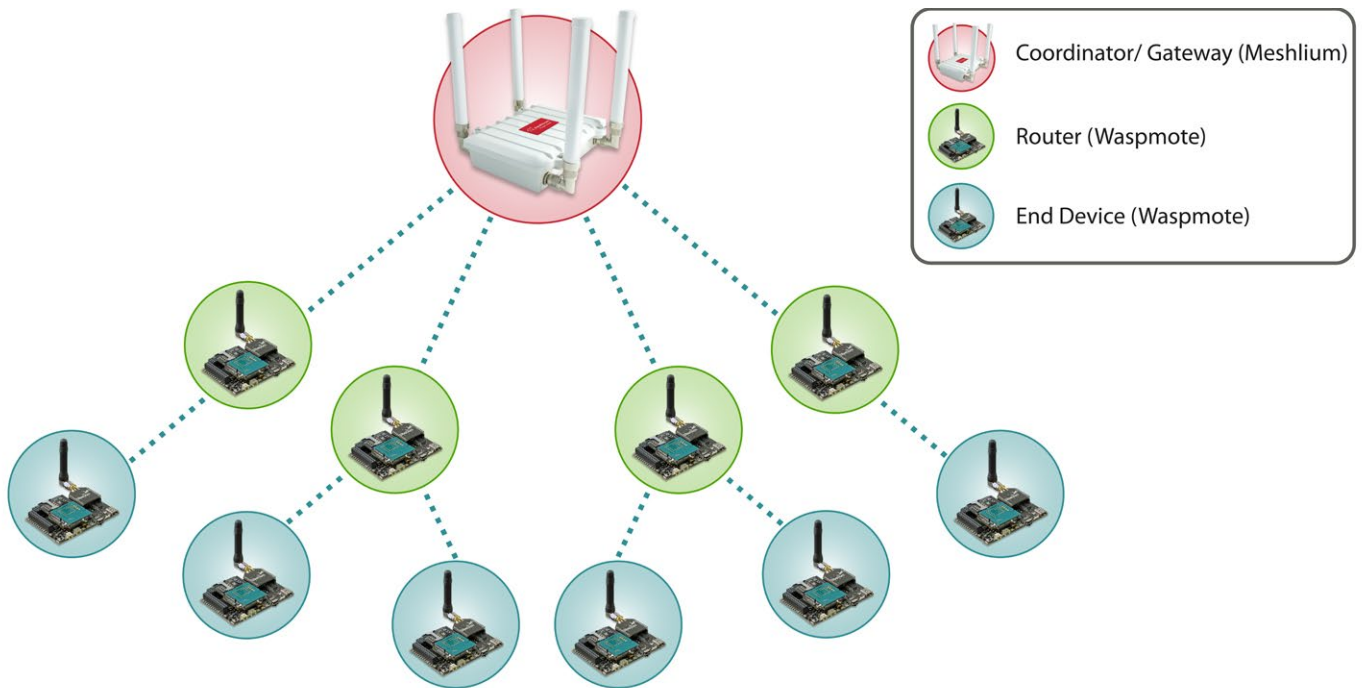
Figure : Tree topology

Regarding the energy section, the transmission power cannot be adjusted, because it is always set to 17 dBm.

2. General Considerations

2.1. Waspmote Libraries

2.1.1. Waspmote XBee Files

WaspXBeeCore.h, WaspXBeeCore.cpp, WaspXBeeZB.h, WaspXBeeZB.cpp

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

```
#include <WaspXBeeZB.h>
```

2.1.2. Constructor

To start using Waspmote XBee library, an object from class 'WaspXBeeZB' must be created. This object, called 'xbeeZB', is created inside 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

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

Related Variables

```
xbeeZB.sourceMacHigh[0-3]  → stores the 32 upper bits of MAC address
xbeeZB.sourceMacLow [0-3]  → stores the 32 lower bits of MAC address
```

When returning from 'xbeeZB.getOwnMacLow' the related variable 'xbeeZB.sourceMacLow' will be filled with the appropriate values. Before calling the function, the related variable is created but it is empty. Almost every function has a related variable, and it will be indicated when the function was explained.

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 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 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

```
{
  xbeeZB.writeValues(); // Keep values after rebooting
}
```

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

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

Related Variables

> `xbeeZB.commandAT[0-99]` → stores the response given by the module up to 100 bytes

• Send command AT example:

> http://www.libelium.com/development/waspmote/examples/zb-13-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 300) 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.

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

The `ON()` function 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 (ZIGBEE 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
- **timeEnergyChannel**: specifies the time the channels will be scanned
- **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
- **parentNA**: specifies the address of the parent
- **deviceType**: specifies the type of the device
- **extendedPAN**: specifies the extended PAN ID
- **maxUnicastHops**: specifies the maximum number of hops in an Unicast transmission
- **maxBroadcastHops**: specifies the maximum number of hops in a Broadcast transmission
- **stackProfile**: specifies the stack of ZigBee protocol used
- **joinTime**: specifies the time the Coordinator allows nodes joining the network
- **channelVerification**: specifies if the router needs a Coordinator to maintain in the network
- **joinNotification**: specifies is a message is sent when a node joins the network
- **aggregateNotification**: specifies the time between consecutive aggregate route broadcast messages
- **encryptOptions**: specifies the options for encryption mode
- **networkKey**: specifies the network key
- **powerMode**: specifies the power mode used

Example of use

```
{
    xbeeZB.ON();
}
```

Expansion Radio Board (XBee ZigBee)

The Expansion Board allows to connect two communication modules at the same time in the Waspmote sensor platform. This means a lot of different combinations are possible using any of the wireless radios available for Waspmote: 802.15.4, ZigBee, DigiMesh, 868 MHz, 900 MHz, LoRaWAN, LoRa, Sigfox, Bluetooth Pro, Bluetooth Low Energy, RFID/NFC, WiFi, GPRS Pro, GPRS+GPS and 3G/GPRS. Besides, the following Industrial Protocols modules are available: RS-485/Modbus, RS-232 Serial/Modbus and CAN Bus.

Some of the possible combinations are:

- LoRaWAN - GPRS
- 802.15.4 - Sigfox
- 868 MHz - RS-485
- RS-232 - WiFi
- DigiMesh - 3G/GPRS
- RS-232 - RFID/NFC
- WiFi - 3G/GPRS
- CAN bus - Bluetooth
- etc.

Remark: *GPRS Pro, GPRS+GPS and 3G/GPRS modules do not need the Expansion Board to be connected to Waspmote. They can be plugged directly in the socket1.*

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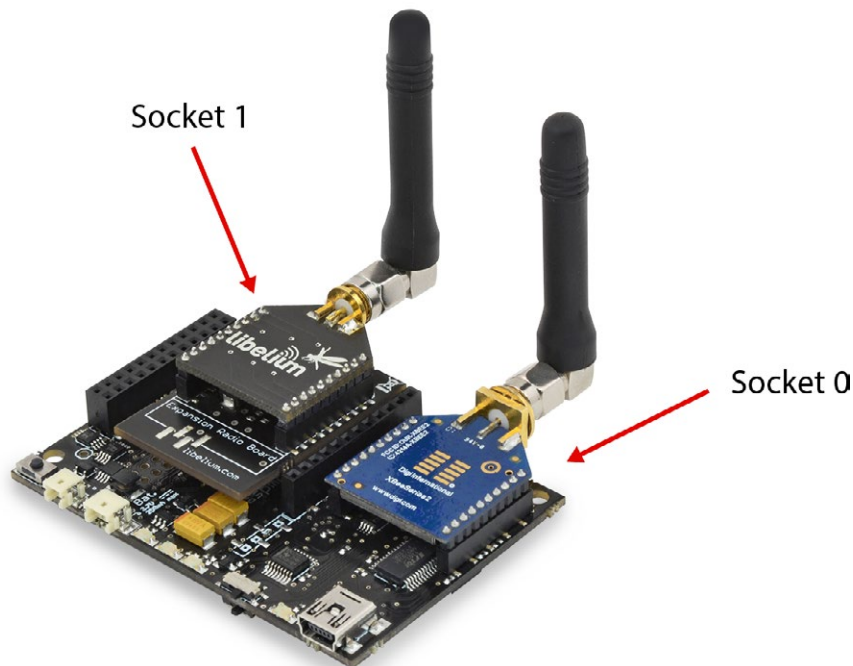
The API provides a function called 'ON' in order to switch the XBee module on. This function supports a parameter which permits to select the SOCKET. It is possible to choose between SOCKET0 and SOCKET1.

Selecting SOCKET0 (both are valid):

```
xbeeZB.ON();
xbeeZB.ON(SOCKET0);
```

Selecting SOCKET1:

```
xbeeZB.ON(SOCKET1);
```

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

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

```
WaspXBeeZB xbeeZB_2 = WaspXBeeZB();
```

Finally, it is necessary to initialize both modules. For example, xbeeZB is initialized in SOCKET0 and xbeeZB_2 in SOCKET1 as follows:

```
xbeeZB.ON(SOCKET0);
xbeeZB_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

The OFF() function closes the UART and switches the XBee OFF.

Example of use

```
{
  xbeeZB.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. Node Types

ZigBee defines three different device types: coordinator, router, and end devices. An example of a possible network is shown below:
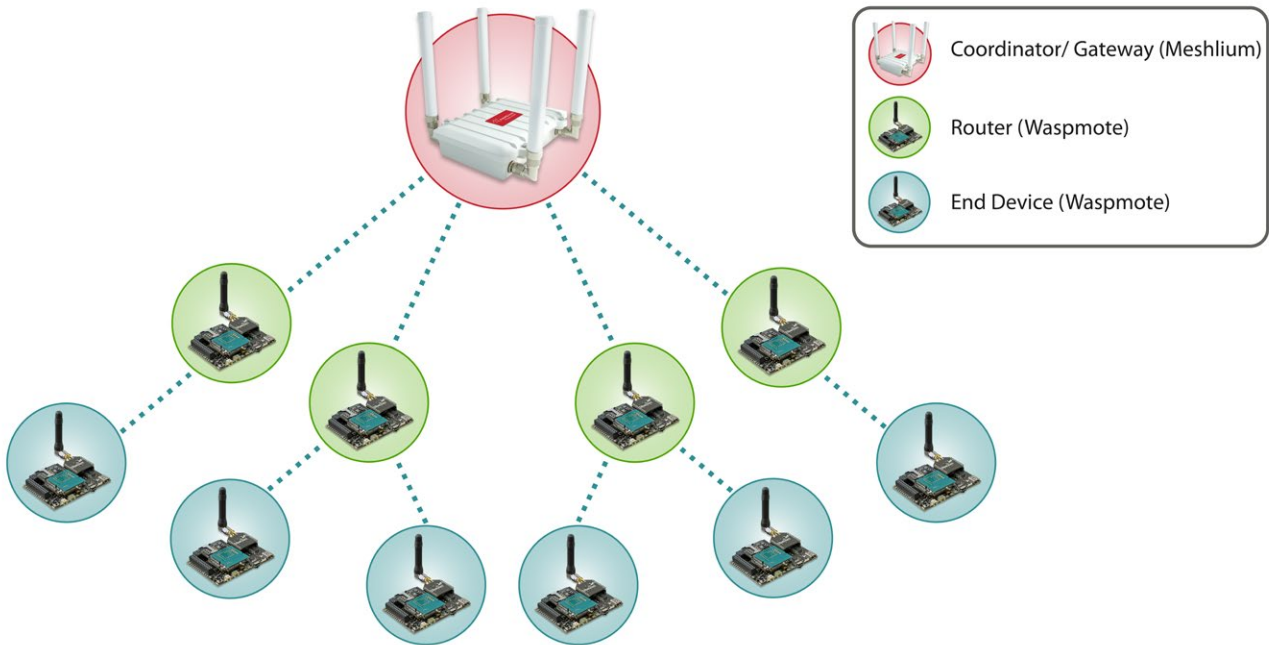
Figure : Tree topology

Coordinator

Each ZigBee network must have one coordinator. A coordinator has the following characteristics:

- It selects the channel and PAN ID (both 64-bit and 16-bit) to start the network.
- It can allow routers and end devices to join the network.
- It can assist in routing data.
- It can not sleep. It has to be always awake.

Router

A router has the following characteristics:

- It must join a ZigBee network before it can transmit, receive or route data.
- After joining, it can allow routers and end devices to join the network.
- After joining, it can route data.
- It can not sleep. It has to be always awake.

End Device

An End Device has the following characteristics:

- It must join a ZigBee network before it can transmit or receive data.
- It can not allow devices to join the network.
- It must always transmit and receive RF data through its parent.
- It can not route data.
- It can sleep.

In ZigBee networks, the coordinator must select a PAN ID (64-bit and 16-bit) and channel to start a network. After that, it behaves essentially like a router. The coordinator and routers can allow other devices to join the network and route data.

After an end device joins a router or coordinator, it must be able to transmit or receive RF data through that router or coordinator. The router or coordinator that allowed the end device to join becomes its parent. Since the end device can sleep, the parent must be able to buffer or retain incoming data packets destined for the end device until it is able to wake up and receive the data.

4.2. MAC Address

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. Due to ZigBee uses 16-bit addresses to send packets inside a network, MAC address is not as much important as in 802.15.4 modules.

Example of use:

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

Related Variables

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

Besides, XBee modules provide a sticker on the bottom side where the MAC address is indicated. MAC addresses are specified as 0013A200xxxxxxxx.

Figure : MAC adress

4.3. PAN ID Parameters

Firstly, it is important to keep in mind different PAN ID parameters:

*   **Extended PAN ID:** This is a *readable/writable* 64-bit parameter which refers to the PAN ID the XBee module wants to join in the case of routers and end devices. In the case of coordinators, it will set the new operating 64-bit PAN ID.
*   **Operating Extended PAN ID:** This is a *read-only* 64-bit parameter which refers to the 64-bit PAN ID the XBee module is working on.
*   **Operating 16-bit PAN ID:** This is a *read-only* 16-bit parameter which refers to the 16-bit PAN ID the XBee module is working on.

Each ZigBee network is defined by both 64-bit and 16-bit operating PAN IDs. Devices on the same ZigBee network must share the same 64-bit and 16-bit PAN IDs.

ZigBee routers and end devices should be configured with the operating 64-bit PAN ID the coordinator is working on. If a joining node has set its PAN ID, it will only join a network with that operating 64-bit PAN ID. They acquire the 16-bit PAN ID from the coordinator when they join a network.

4.3.1. Extended PAN ID

64-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.

There are two possibilities to set:

*   **Non-zero PAN ID:** If the Coordinator sets this parameter to a non-zero value, this PAN ID will be set as operating 64-bit PAN ID. If Routers and End Devices set a non-zero value, they will only join a network with that PAN ID.
*   **Zero PAN ID:** If the Coordinator sets this parameter to a zero value, it will select a random operating 64-bit PAN ID. Routers and End Devices will attempt to join this random network if they are not associated to any network yet.

Example of use

```
{
  uint8_t  panid[8]={0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
  xbeeZB.setPAN(panid);
  xbeeZB.getPAN();
}
```

Related Variables

xbeeZB.PAN_ID[0-7] → stores the 64-bit Extended PAN ID

4.3.2. Operating Extended PAN ID

The Operating Extended PAN ID is the current 64-bit PAN ID the module is working on. The operating Extended PAN ID is intended to be a unique, non-duplicated value.

When a coordinator starts a network, it can either start a network on a specific configured Extended PAN ID, or it can select a random PAN ID by previously setting the Extended PAN ID to zero.

In the case of Routers and End Devices, they will only join the non-zero Extended PAN ID they have set. If Extended PAN ID has been set to zero, they will try to join any network.

Example of use:

```
{
  xbeeZB.getOperating64PAN();
}
```

Related Variables

xbeeZB.operating64PAN[0-7] → stores the 64-bit Operating PAN ID

4.3.3. Operating 16-bit PAN ID

The Operating 16-bit PAN ID is the current 16-bit PAN ID the module is working on. The Operating 16-bit PAN ID is used as a MAC layer addressing field in all RF data transmissions between devices in a network. It is used to reduce overheads on application level tasks.

The Operating 16-bit PAN ID is set by the coordinator when the network is created and it is set to routers and end devices when they join the network. Thus, the 16-bit PAN ID can not be configured by the user.

Example of use

```
{
  xbeeZB.getOperating16PAN ();
}
```

Related Variables

> `xbeeZB.operating16PAN[0-1]` → stores the 16-bit Operating PAN ID

4.4. Network Address

16-bit Network Address. When a node joins the network, the Coordinator assigns a 16-bit address that can not be changed until the node leaves the PAN or the Coordinator decides to change it. ZigBee uses 16-bit addresses to send the packets inside the network to reduce overhead. If the 16-bit address of the destination node is not known, a process called 'Address Discovery' will be executed. This process is transparent to the user.

Example of use

```
{
    xbeeZB.getOwnNetAddress();
}
```

Related Variables

> `xbeeZB.sourceNA[0-1]` → stores the 16-bit network address

4.5. 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 application level. It is also used to search a node using its NI.

Example of use

```
{
    xbeeZB.setNodeIdentifier("node01");
    xbeeZB.getNodeIdentifier();
}
```
Related Variables

> `xbeeZB.nodeID[0-19]` → stores the 20-byte max string Node Identifier

4.6. Channel

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

ZigBee works on 2.4GHz band, using 16 channels. Channel is selected by the Coordinator when starting the network, and it can not be modified.
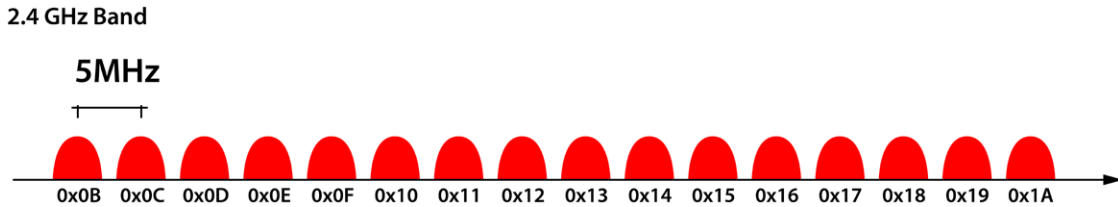
Figure : Operating Frequency Bands

Channel Number	Frequency	Supported by
0x0B – Channel 11	2,400 – 2,405 GHz	PRO
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
0x18 – Channel 24	2,465 – 2,470 GHz	PRO

Figure : Channel Frequency Numbers

Example of use:

```
{
  xbeeZB.getChannel();
}
```

Related Variables

xbeeZB.channel → stores the operating channel

4.7. Device Type Identifier

Device type value. This value can be used to differentiate multiple XBee-based products.

Example of use

```
{
  uint8_t devicetype[4]={0x01, 0x02, 0x03, 0x04};
  xbeeZB.setDeviceType(devicetype);
  xbeeZB.getDeviceType();
}
```

Related Variables

xbeeZB.deviceType[0-3] → stores the device type identifier

5. Power Gain and Sensibility

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

5.1. Power Level

Power level (dBm) at which the RF module transmits conducted power. Its only possible value is +17 dBm (parameter = 4).

Note: *XBee-PRO maximum value is +17dBm and XBee-PRO(International) is +10dBm.*

Note 2: *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:

$$mW = 10^{(value\ dBm/10)}$$

Example of use

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

Related Variables

> `xbeeZB.powerLevel` → stores the power output level selected

Power Level configuration example:

> http://www.libelium.com/development/waspmote/examples/zb-14-setread-power-level

5.2. 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

```
{
  xbeeZB.getRSSI();
}
```

Related Variables

> `xbeeZB.valueRSSI` → stores the RSSI of the last received packet

The ideal working mode is getting 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/zb-06-get-rssi

6. Radio Channels

XBee ZigBee module works on 2.4GHz band, so there are 14 channels that can be used. These channels are explained in Chapter 4.

When starting a network, the Coordinator performs an energy scan to find the channel with less energy. When joining a network, the router or end device performs an active scan to find any coordinator or router available to join. There are some parameters involved in this channel election.

6.1. Scan Channels

List of channels to scan. If it is a Coordinator, the list of channels to choose from prior to starting the network. If its a router/end device, the list of channels to scan to find a Coordinator/Router to join the network. Setting this parameter to 0xFFFF, all the channels will be scanned.

Example of use

```
{
  xbeeZB.setScanningChannels(0xFF,0xFF);
  xbeeZB.getScanningChannels();
}
```

Related Variables

> `xbeeZB.scanChannels[0-1]` → stores the list of channels to scan

6.2. Scan Duration

Scan duration exponent. If it is a Coordinator, duration of the Active and Energy Scans (on each channel) that are used to determine an acceptable channel and PAN ID for the Coordinator to startup on. If it is a Router/End Device, duration of Active Scan (on each channel) used to locate an available Coordinator/Router to join during Association.

Scan time is measured as: (channels to scan) * (2^SD) * 15,36ms.

Example of use:

```
{
  xbeeZB.setDurationEnergyChannels(3); // Set exponent: scan time=channels*122,88ms
  xbeeZB.getDurationEnergyChannels(); // Get exponent
}
```

Related Variables

> `xbeeZB.timeEnergyChannel` → stores the exponent

7. Networking methods

7.1. Topologies

ZigBee provides different topologies 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.
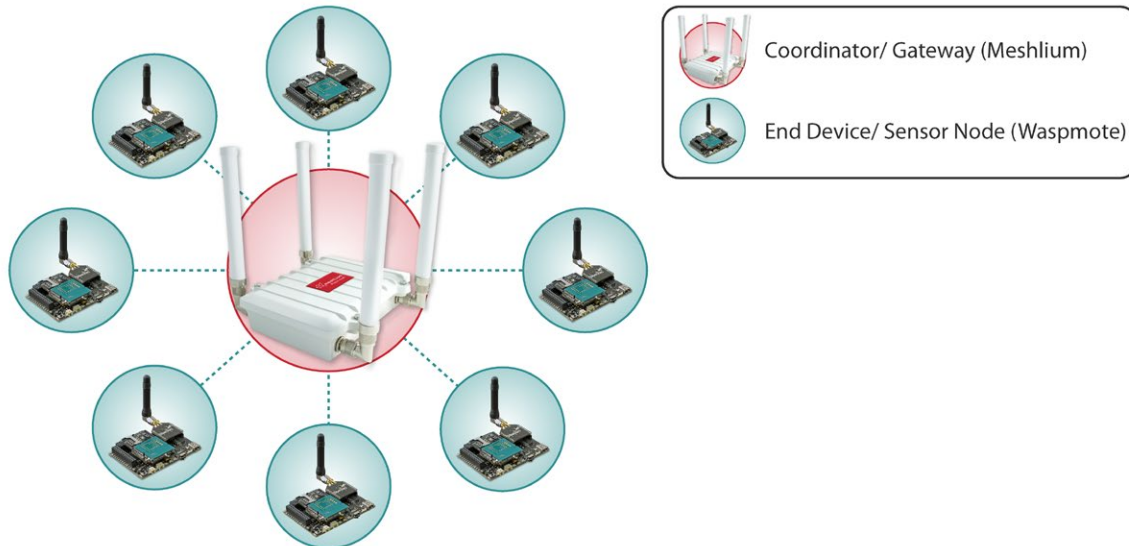
Figure : Star Topology

- **Tree**: a tree network has a top node with a branch/leaf structure below. To reach its destination, a message travels up the tree (as far as necessary) and then down the tree.
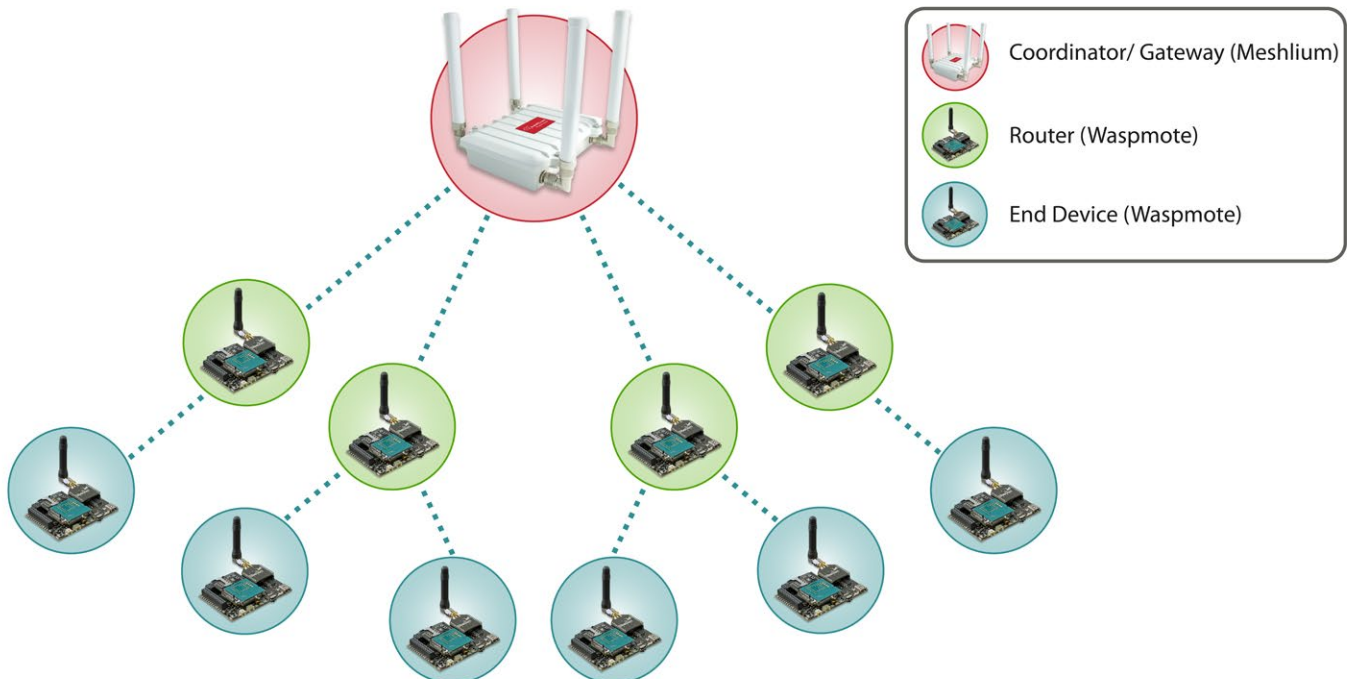
Figure : Tree Topology

ZigBee is aimed at working on star or tree topology. Peer-to-Peer topologies does not have sense since only End Devices can sleep. Mesh topologies may be found interesting, but due to only End Devices can sleep, a real mesh can not be done. To work on a pure mesh topology, it is recommended to choose another protocol as DigiMesh.

7.2. ZigBee Architecture

There are some concepts that will help to understand ZigBee architecture.
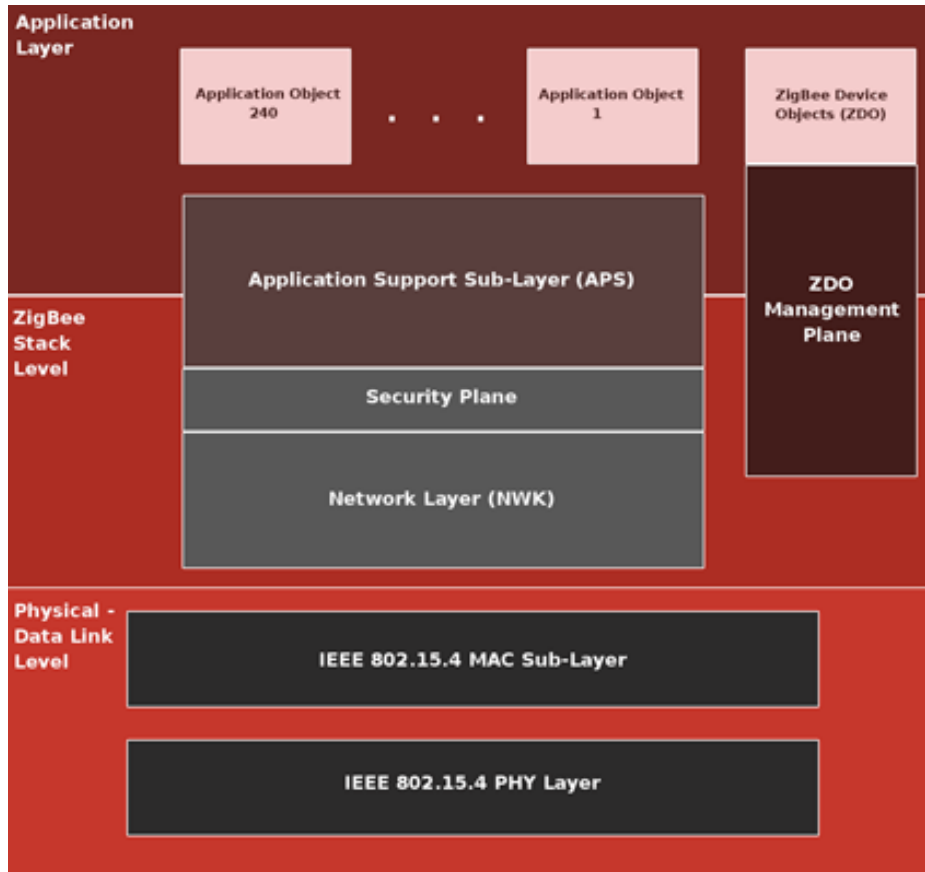
Figure : ZigBee Stack

7.2.1. ZigBee Device Objects

ZigBee Device Object (ZDO), a protocol in the ZigBee protocol stack, is responsible for overall device management, and security keys and policies. The ZDO is like an special application object that is resident on all ZigBee nodes. ZDO has its own profile, known as the ZigBee Device Profile (ZDP), where the application end points and other ZigBee nodes can access. ZDO represents the ZigBee node type of the device and has a number of initialization and communication roles.

7.2.2. ZDO Management Plane

This plane spans the Application Support Sub-layer (APS) and NWK layers, and allows ZDO to communicate with these layers when performing its internal tasks. It also allows the ZDO to deal with requests from applications for network access and security functions using ZDP messages.

7.2.3. Application Support Sub-layer (APS)

The APS is responsible of communicating with the relevant application using the endpoint information of the message. The message is passed through the Service Access point (SAP) which exists between the APS layer and each application. It is also responsible of maintaining binding tables.

7.2.4. Network Layer

It handles network addressing and routing invoking actions in the MAC layer. It is controlled using API functions like `getOwnMac()`, `setOwnNetAddress()` or `getOwnNetAddress()`.

7.2.5. Security Plane

In addition, a Security Plane spans and interacts with the APS and NWK layers. This provides security services - for example, security key management, data stream encryption and decryption. It may use hardware functions provided in the node to perform the encode and decode functions efficiently. It is controlled using API functions like `setEncryptionMode()`, `setLinkKey()`, `setNetworkKey()` or `setEncryptionOptions()` (see chapter "Security and Data Encryption" for further information).

7.2.6. Application Framework

The Application Framework (AF) contains the application objects and facilitates interaction between the applications and the APS layer. An application object interacts with the APS layer through an interface known as a Service Access Point (SAP).

7.2.7. Service Access Points

A Service Access Point (SAP) implements a set of operations to pass information and commands between layers.

7.2.8. Application Endpoints

A node may have several applications running on it (several sensors) each of which is an application. These application instances on a node are said to be endpoints, where messages can originate and terminate.

In order to route messages arriving to the node to the appropriate application, each application in the node must be uniquely identified and is given an endpoint address. Endpoint addresses for user applications are numbered from 1 to 240. Therefore, to identify a particular application instance in a ZigBee network, the relevant network address and the endpoint address on the node are needed to be supplied.

Endpoint address 255 can be used to broadcast a message to all the user endpoints on a node.

Endpoint address 0 is reserved for a special application called ZDO (ZigBee Device Objects).

7.2.9. Application Profiles

An Application Profile is associated with a particular stack profile and addresses the needs of a specific application or market; for example, the ZigBee Alliance has defined the Home Controls-Lighting (HCL) application profile. It defines a number of devices and functions which are needed or useful for controlling domestic lighting, such as switches, dimmers, occupancy sensors and load controllers (which control the light sources).

The ZigBee stack in a particular network will use the relevant 'Stack Profile' from the ZigBee Alliance. The stack profile determines the type, shape and features of the network, and depends on the field of application, e.g. the Home Controls profile.

More specifically, in each Application Profile a number of Device Descriptions are defined, describing the types of devices the profile supports. For the HCL profile, devices such as Switch Remote Control (a switch), Light Sensor Monochromatic, Dimmer Remote Control, Occupancy Sensor, Switching Load Controller and Dimming Load Controller are available. Each device in an Application Profile has a Device Identifier associated with it.

As well as defining the device types supported, the Application Profile also specifies the information that a device can generate as output and can use as input, together with the format this information takes.

The individual pieces of information are called attributes, and their possible values and format or type are defined as part of the Device Descriptions in the profile. Attributes are grouped together into clusters for the device, which can be either inputs or outputs.

7.2.10. Attributes

Each data item that passes between devices of a ZigBee network is called an attribute. Each attribute has its own unique identifier. For example, a switch device can have an attribute with identifier 'OnOff' whose value represents the action to be performed: On (0xFF), Off (0x00), Toggle (0xF0).

7.2.11. Clusters

A number of attributes are grouped into a cluster, where each cluster has its own unique identifier.

For example, for an HCL Switch Remote Control (SRC) device, there is a cluster with the identifier 'OnOffSRC' containing the attribute OnOff. Clusters may be mandatory or optional for a device to support.

An Application Profile can have several associated clusters. For example, the Home Controls-Lighting (HCL) profile includes the clusters 'OnOffSRC' and 'ProgramSRC' for an SRC device.

The Application Profile defines which clusters are mandatory and which clusters are optional for the device. The clusters supported by a device determine the other devices to which it can communicate. For example, for two devices operating together in temperature monitoring and control, both of them must support compatible clusters concerned with temperature. The sensor device must have an output cluster containing the monitored temperature, and the controller must support an input cluster which can use a temperature to make control decisions.

7.2.12. Binding

At a high level, binding is the process of establishing a relationship between nodes that can communicate in a meaningful way, for example, linking switches with lights. It therefore determines the overall functionality of the network.

The information is exchanged as clusters - in order to bound two applications, they must have compatible clusters. For example, for two applications in different nodes for temperature control, one must be able to generate an output cluster related to temperature, and the other must be able to consume it as an input cluster. The binding between two applications is specified by:

•   The source network address and endpoint of the application where the cluster is generated
•   The destination network address and endpoint of the receiving application
•   The cluster ID of the cluster being sent between them

Bindings are stored in a binding table. This lists the cluster IDs, the network addresses and application endpoints for each association. The types of binding that can be achieved are one-to-one, one-to-many, many-to-one and many-to-many:

•   One-to-one: A binding in which an endpoint is bound to one (and only one) other endpoint .
•   One-to-many: A binding in which a source endpoint is bound to more than one destination endpoint .
•   Many-to-one: A binding in which more than one source endpoint is bound to a single destination endpoint .
•   Many-to-many : A binding in which more than one source endpoint is bound to more than one destination endpoint.

Bindings (the binding entries table) can be stored either on the network Coordinator (indirect binding) or locally on the node generating the source output cluster (direct binding).

7.2.13. Waspmote Application Level

The concepts explained above are used in a special kind of transmission called 'Application Level Addressing'. This kind of transmission is used to send data between two devices using endpoints and clusters.

In order to send data, this transmission uses a packet which is a bit different from the usual .It is explained with an example in chapter 8.6, showing how to include information in the sent packet as Destination Endpoint, Cluster and Profiles.

To receive a packet of this type, an option parameter should be set to notify the module of the incoming special packet. To enable this option:

Example of use:

```
{
  xbeeZB.setAPIoptions(1); // Set explicit frame used in Application Level Addressing
}
```

7.3. ZigBee Addressing

ZigBee supports device addressing and application layer addressing. Device addressing specifies the destination address of the device a packet is destined to. Application layer addressing indicates a particular application recipient, known as a ZigBee endpoint, along with a message type field called a Cluster ID.

7.3.1. Device Addressing

ZigBee protocol specifies two address types: 16-bit and 64-bit addresses.

*   16-bit Network Address : A 16-bit Network Address is assigned to a node when joining a network. The network address is unique to each node in the network, but it may change. ZigBee requires data to be sent to the 16-bit network address of the destination device. This requires the 16-bit network address to be discovered before transmitting the data, executing a process called 'Address Discovery'.
*   64-bit Address : Each node contains a unique 64-bit address. The 64-bit address uniquely identifies a node and is therefore permanent. It is assigned by the manufacturer and it can not be changed.

7.3.2. Application Layer Addressing

The ZigBee application layers define endpoints and cluster identifiers (cluster IDs) that are used to address individual services or applications on a device. An endpoint is a task or application that runs on a ZigBee device, similar to a TCP port. Each ZigBee device may support one or more endpoints. Cluster IDs define a particular function or action on a device. For example: in the ZigBee home controls lighting profile, Cluster IDs would include actions such as "TurnLightOn", "TurnLightOff", "DimLight", etc.

7.4. Connections

All data packets are addressed using both device and application layer addressing fields. Data can be sent as a broadcast, or unicast transmission.

7.4.1. Broadcast

Broadcast transmissions within the ZigBee protocol are intended to be propagated throughout the entire network such that all nodes receive the transmission. To accomplish this, all devices that receive a broadcast transmission will retransmit the packet 3 times.

Each node that transmits the broadcast will also create an entry in a local broadcast transmission table. This entry is used to keep track of each received broadcast packet to ensure the packets are not endlessly transmitted. For each broadcast transmission, the ZigBee stack must reserve buffer space for a copy of the data packet. Since broadcast transmissions are retransmitted by each device in the network, broadcast messages should be used sparingly.

Broadcast transmissions are sent using a 64-bit address of 0x000000000000FFFF. Any RF module in the PAN will accept a packet that contains a broadcast address. When configured to operate in Broadcast Mode, receiving modules do not send ACKs (Acknowledgments).

See examples in chapter "Connectivity" → Sending data for further information.

7.4.2. Unicast

Unicast ZigBee transmissions are always addressed to the fact that 16-bit address of the destination device. However, only the 64-bit address of a device is permanent due to the 16-bit address can change. Therefore, ZigBee devices may employ network address discovery to identify the current 16-bit address that corresponds to a known 64-bit address, and route discovery to establish a route.

See examples in chapter 8.6 for further information.

- **Network Address Discovery**

  Data transmissions are always sent to the 16-bit network address of the destination device. However, since the 64-bit address is unique to each device and is generally known, ZigBee devices must discover the network address that was assigned to a particular device when it joined the PAN before they can transmit data. To do this, the device initiating a transmission sends a broadcast network address discovery transmission throughout the network. This packet contains the 64-bit address of the device the initiator needs to send data to. Devices that receive this broadcast transmission check to see if their 64-bit address matches the 64-bit address contained in the broadcast transmission. If the addresses match, the device sends a response packet back to the initiator, supplying the network address of the device with the matching 64-bit address. When this response is received, the initiator can then transmit data.

- **Route Discovery**

  ZigBee employs mesh routing to establish a route between the source device and the destination. Mesh routing allows data packets to traverse multiple nodes (hops) in a network to route data from the source to its destination. Routers and coordinators can participate in establishing routes between source and destination devices using a process called route discovery. The Route discovery process is based on the AODV (Ad-hoc On demand Distance Vector routing) protocol.

- **Retries and ACKs**

  ZigBee includes acknowledgment packets at both the Mac and Application Support (APS) layers. When data is transmitted to a remote device, it may traverse multiple hops to reach the destination. As data is transmitted from one node to its neighbour, an acknowledgment packet (ACK) is transmitted in the opposite direction to indicate that the transmission was successfully received. If the ACK is not received, the transmitting device will retransmit the data up to 4 times. This ACK is called the Mac layer acknowledgment.

  In addition, the device that originated the transmission expects to receive an acknowledgment packet (ACK) from the destination device. This ACK will traverse the same path that the data traversed, but in the opposite direction. If the originator fails to receive this ACK, it will retransmit the data, up to 2 times until an ACK is received. This ACK is called the ZigBee APS layer acknowledgment.

7.4.3. Many to one

Since ZigBee unicast transmissions may require some combination of broadcast network address discovery and/or route discovery, having large numbers of devices unicasting data to a single gateway or collector device may not be the best solution for large networks.

To work around this potential problem, ZigBee includes provisions to support many-to-one transmissions, where many devices in a network can all transmit data to a central gateway or collector device without causing a flood of route discoveries. To accomplish this, the collector device sends a periodic broadcast transmission identifying itself as a collector device. All other devices that receive this broadcast transmission create a reverse routing table entry back to the collector. When the remote devices transmit data to the collector, they first transmit a route record frame (that records the entire route from the remote to the collector) before transmitting the data. The route record frame provides the collector with the entire route to each remote it receives data from. The collector can use the information in the route record frames to store return routes. This process effectively establishes routes between the collector and all devices in the network using a single broadcast transmission instead of many route discoveries.

7.4.4. Parameters involved

There are some parameters involved in these explained connections

- **Maximum Unicast Hops**

  It sets the maximum unicast hops limit and the unicast timeout. The timeout is computed as: (50*NH)+100ms. The default value of 0x1E (1,6seconds) is enough to traverse around 8 hops.

Example of use:

```
{
  xbeeZB.setMaxUnicastHops(0x03); // Set max unicast hops limit
  xbeeZB.getMaxUnicastHops(); // Get max unicast hops limit
}
```

Related Variables

xbeeZB.maxUnicastHops → stores the max unicast hops limit

- **Maximum Broadcast Hops**

    It sets the maximum number of hops for each broadcast data transmission.

Example of use:

```
{
  xbeeZB.setMaxBroadcastHops(0x10); // Set max broadcast hops limit
  xbeeZB.getMaxBroadcastHops(); // Get max broadcast hops limit
}
```

Related Variables

> xbeeZB.maxBroadcastHops → stores the max broadcast hops limit

- **Aggregate Routing Notification (Many to one)**

    Time between consecutive aggregate route broadcast messages. Using this parameter, many to one transmissions will be enabled.

Example of use:

```
{
  xbeeZB.setAggregateNotification(0x20); // Set time between aggregate messages
  xbeeZB.getAggregateNotification (); // Get time between aggregate messages
}
```

Related Variables

> xbeeZB.aggregateNotification → stores the time between consecutive messages

7.5. Maximum payloads

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

	Unicast	Broadcast
Encrypted	66Bytes	84Bytes
Un-Encrypted	74Bytes	92Bytes

Figure : Maximum Payloads Size

7.5.1. Maximum Payload

If the maximum payload is not known when operating in an application, it can be discovered in real time.

Example of use

```
{
    xbeeZB.getPayloadBytes();
}
```

Related Variables

> xbeeZB.maxPayloadBytes[0-1] → stores the maximum payload is available at the moment

7.6. Sending Data

7.6.1. Using Waspmote Frame

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/data-frame-guide/

7.6.2. Sending function

The function `send()` sends a packet via XBee module.

Firstly, the **destination address** must be defined depending on the addressing mode:

- Define 64-bit addressing unicast mode (must specify the destination MAC address). For example:

  ```
  {
      char rx_address[] = "0013A2004030F6BC";
  }
  ```

- Define broadcast mode:

  ```
  {
      char rx_address[] = "000000000000FFFF";
  }
  ```

Finally, there are different **sending function** prototypes depending on the data sent. It is possible to send text messages or binary data:

- Send strings:

  ```
  {
      char data[] = "this_is_the_data_field";
      xbeeZB.send( rx_address, data);
  }
  ```
- Send Waspmote Frames:

  ```
  {
      xbeeZB.send( rx_address, frame.buffer, frame.length );
  }
  ```
- Send Array of bytes (it is mandatory to specify the length of the data field):

  ```
  {
      uint8_t  data[5] = {0x00, 0x01, 0x54, 0x76, 0x23};
      xbeeZB.send( rx_address, data, 5);
  }
  ```

The sending function implements application-level retries. By default, up to 3 retries are done in the case the sending process fails. If a different number of maximum retries is needed, the `setSendingRetries()` function permits to do it. This function changes the value of the API variable. When a new `send()` function is called, the new maximum number of retries will be used.

Keep in mind that using a high number of retries could lead to a longer execution time of the `send()` function, which means more power consumption on Waspmote and less channel availability for the rest of network nodes. Probably, after 3 or 4 (failed) retries, it does not make sense to keep on trying.

Parameter range: From 0 to 10
Default: 3

Example of use:

```
{
    xbeeZB.setSendingRetries(10);
}
```

Related variables:

> `xbeeZB._send_retries` → stores the maximum number of application-level retries

7.6.3. Examples

- Send packets in unicast mode:
    http://www.libelium.com/development/waspmote/examples/zb-03-send-packets

- Send packets in broadcast mode:
    http://www.libelium.com/development/waspmote/examples/zb-05a-send-broadcast

- Send packets using the expansion board:
    http://www.libelium.com/development/waspmote/examples/zb-07a-expansion-board-send

- Complete example, send packets in unicast mode and wait for a response:
    http://www.libelium.com/development/waspmote/examples/zb-09a-complete-example-send

7.7. Receiving Data

7.7.1. Receiving function

The function `receivePacketTimeout()` waits a period of time trying to receive a packet through the XBee module. The period of time to wait is specified in millisecond units as input when calling the function.

The Waspmote API defines the following variables to store information from the received packets:

Variable	Description
`uint8_t   _payload[MAX_DATA]`	Buffer to store the received packet
`uint16_t  _length`	Specifies the length of the buffer contents
`uint8_t   _srcMAC[8]`	Specifies the source MAC address when a packet is received

When this function is called, several answers might be expected:

'0' → OK: The command has been executed with no errors

'1' → Error: timeout when receiving answer

'2' → Error: Frame Type is not valid

'3' → Error: Checksum byte is not available

'4' → Error: Checksum is not correct

'5' → Error: Error escaping character in checksum byte

'6' → Error: Error escaping character within payload bytes

'7' → Error: Buffer full. not enough memory space

Example of use:

```
{
   uint8_t  error;
   error = xbeeZB.receivePacketTimeout( 10000 );
}
```

Related variables:

xbeeZB._payload[] → Buffer where the received packet is stored

xbeeZB._length → Length of the buffer

xbeeZB._srcMAC[0-7] → Source's MAC address

7.7.2. Examples

- Receiving packets example:
  http://www.libelium.com/development/waspmote/examples/zb-04-receive-packets

- Receive packets in broadcast mode (the same procedure as if it was unicast mode):
  http://www.libelium.com/development/waspmote/examples/zb-05b-receive-broadcast

- Receive packets using the expansion board:
  http://www.libelium.com/development/waspmote/examples/zb-07b-expansion-board-reception

- Complete example, receive packets and send a response back to the sender:
  http://www.libelium.com/development/waspmote/examples/zb-09b-complete-example-receive

8. Starting a Network

8.1. Coordinator Startup

To create a ZigBee network, a coordinator must be started on a channel and PAN ID (64-bit and 16-bit). Once the coordinator has started, routers and end devices can join the network.

Some parameters explained in previously chapters as 'Extended PAN ID', 'Scan Channels' and 'Scan Duration' are used in network startup.

To select a correct channel and PAN ID, the coordinator performs an energy scan and PAN scan. 'Scan Channels' and 'Scan Duration' are used to determine the duration and the list of channels to scan. 'Extended PAN ID' is used to select an ID different from zero.

After completing the energy and PAN scans, the coordinator uses the results to select the channel with less energy and a PAN ID unused in other PAN. If security should be set at startup, please see chapter "Synchronizing the Network".

After the coordinator has successfully started, it behaves similar to a router - it can allow devices to join the network, and it can transmit, route and receive data. The coordinator saves the selected channel and PAN ID settings into non-volatile memory so this information will persist through power cycles.

If a coordinator has successfully started a PAN and other coordinator is powered, there will be two PANs. It will perform the explained scans and will choose a channel and PAN ID to work on. A coordinator can't be part of a network if another coordinator already exists.

• Coordinator creates a network example:

  http://www.libelium.com/development/waspmote/examples/zb-01a-coordinator-creates-network
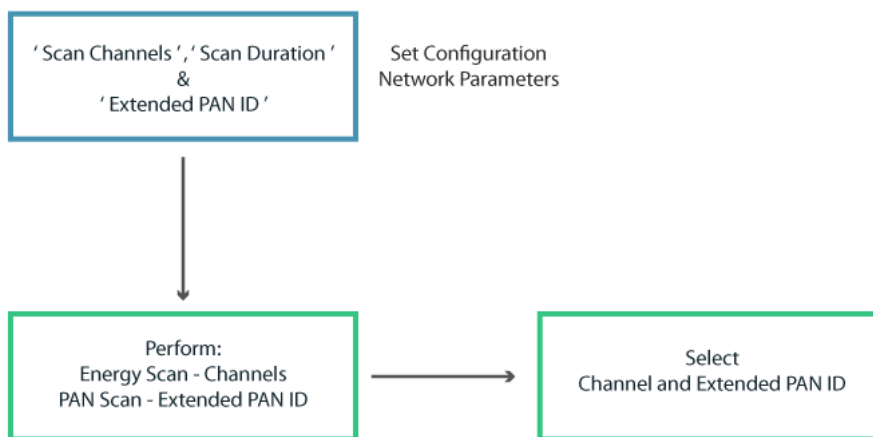

Figure : Coordinator Startup

8.2. Coordinator wake up process from Reset

When a coordinator wakes up from a reset or power cycle, its checks its operating channel and PAN ID against 'Scan Channels' and 'Extended PAN ID' parameters. It also verifies the security policy. If some configuration parameters do not match the saved channel, PAN ID or security, it will leave the PAN and will start in other PAN matching the configuration parameters.

• Coordinator resets network example:

> http://www.libelium.com/development/waspmote/examples/zb-01b-coordinator-resets-network

Figure : Coordinator Wake Up Process

8.3. Related Parameters

There are some parameters involved in a starting a network process.

8.3.1. Node Join Time

It specifies the time a Coordinator or Router allows a node to join the network. This value can be changed at run time without requiring a Coordinator or Router to restart. This time starts once the Coordinator or Router has waken up. The timer is reset on power-cycle or when 'Node Join Time' changes.

Example of use:

```
{
  xbeeZB.setJoinTime(0xFF); // Set Join Time : always allows joining
  xbeeZB.getJoinTime(); // Get Join Time
}
```

Related Variables

> xbeeZB.joinTime → stores the join time selected

8.3.2. ZigBee Stack Profile

It specifies the stack profile value. It must be the same in all devices that may join the network. Available values are:

- 0 : Private Network.
- 1 : ZigBee 2006.
- 2 : ZigBee-Pro 2007

Example of use:

```
{
  xbeeZB.setStackProfile(2); // Set ZigBee stack profile to ZB 2007
  xbeeZB.getStackProfile(); // Get ZigBee stack profile
}
```

Related Variables

> xbeeZB.stackProfile → stores the ZigBee stack profile

9. Joining an Existing Network

Before a router or end device can join a ZigBee network, it must locate a nearby coordinator or another router that has already joined to join through it.

9.1. Router Joining a Network

If a router has not joined a network, it performs a PAN scan in all the channels specified in the list 'Scan Channels', looking for a coordinator or router operating with a valid PAN ID that is allowing joins. The router must find a device that meets the following parameters:

• The 'operating 64-bit PAN ID' of the device is valid depending on the router 'PAN ID'. If router 'PAN ID' is set to zero, any value will be valid. If not, it will be the same to be a valid value.
• The discovered device is allowing joins. Parameter 'Node Join Time' has not expired.
• The discovered device is operating in one of 'Scan Channels' channels.
• The discovered device is operating with the same ZigBee stack profile.
• The security police is the same in both devices.

If a device is discovered during the PAN scan and meets all the above requirements, the joining device will send a join request to the device and attempt to join the PAN. If the joining device receives a join response, then the device is considered joined to the PAN.

The status of the last PAN scan and join attempt is recorded into the 'Association Indication' parameter.

9.1.1. Examples

• Router joins known network:

> http://www.libelium.com/development/waspmote/examples/zb-02a-router-joins-known-network

• Router joins unknown network:

> http://www.libelium.com/development/waspmote/examples/zb-02b-router-joins-unknown-network

9.2. End Device Joining a Network

If an End Device has not joined a network, it performs a PAN scan in all the channels specified in the list 'Scan Channels', looking for a coordinator or router operating in a valid PAN ID that is allowing joins. The End Device must find a device that meets the following parameters:

• The 'operating 64-bit PAN ID' of the device is valid depending on the End Device's 'PAN ID'. If End Device's 'PAN ID' is set to zero, any value will be valid. If not, it will be the same to be a valid value.
• The discovered device is allowing joins. Parameter 'Node Join Time' has not expired.
• The discovered device has room for at least one more node. 'Number of Remaining Children' parameter.
• The discovered device is operating in one of 'Scan Channels' channels.
• The discovered device is operating with the same ZigBee stack profile.
• The security police is the same in both devices.

If a device is discovered during the PAN scan meets all the above requirements, the joining device will send a join request to the device and attempt to join the PAN. If the joining device receives a join response, then the device is considered joined to the PAN.

The status of the last PAN scan and join attempt is recorded into the ' Association Indication' parameter.

9.3. Router / End Device Wake Up from Reset Process

Once a router or end device has joined a PAN ID and channel, it retains that information through power cycle or reset events. When the router or end device wakes up from a reset or power cycle, it checks its operating channel and PAN ID against the network configuration commands 'Scan Channels' and 'Extended PAN ID'. It also verifies the applied security policy against the security configuration. If device's operating channel, PAN ID, or security policy does not match the configuration commands, it will leave its current channel and PAN ID and attempt to join a new network based on its 'Scan Channels', 'Extended PAN ID' and security parameter values.

Figure : Router / End Device Coming up from Reset

9.4. Channel Verification

Channel verification behaviour is supported for both routers and end devices through the 'Channel Verification' parameter. If channel verification is enabled, routers will communicate with the coordinator and end devices will monitor their communications with their parent to determine if they are on a valid channel.

9.4.1. Router Channel Verification

When a router wakes up from a power cycle, if 'Channel Verification' is enabled, the router will send a 64-bit address discovery transmission to the coordinator to discover its 64-bit address. If the coordinator does not respond after 3 request attempts, the router will leave the current network and attempt to join a new network based on its network configuration parameters.

9.4.2. End Device Channel Verification

If 'Channel Verification' is enabled, the end device tracks the status of its poll requests with its parent. If the end device's parent does not send an acknowledgment for 3 consecutive poll requests, the end device will leave its current channel and PAN and attempt to join a network based on its network configuration parameters.

9.5. Related Parameters

There are some parameters related to a joining a network process.

9.5.1. Association Indication

Specifies the status of the last PAN scan and join attempt of the module. Possible status are:

- 0x00: Successful. Coordinator started or Router/End Device joined with a parent.
- 0xAB: Attempted to join a device did not respond.
- 0xAC: Secure join error. Network security key received unsecured.
- 0xAD: Secure join error. Network security key not received.
- 0xAF: Secure join error. Joining device does not have the right preconfigured link key.
- 0x21: Scan found no PANs.
- 0x22: Scan found no valid PANs based on current 'Scan Channels' and 'Extended PAN ID'.
- 0x23: Valid Coordinator or Routers found, but they are not allowing joining.
- 0x27: Node joining attempt failed.
- 0x2A: Coordinator Start attempt failed.
- 0xFF: Scanning for a parent.
- 0x2B: Checking for an existing coordinator.

Example of use:

```
{
  xbeeZB.getAssociationIndication();
}
```

Related Variables

xbeeZB.associationIndication → stores the Association Indication Status

9.5.2. Number of Remaining Children

Specifies the number of End Devices that can join a coordinator or router. Its maximum value for Coordinators is 10 and 12 for a Router. When a Router joins, it doesn´t count as a children, so as many routers as wanted can join a network.

Example of use:

```
{
  xbeeZB.getRemainingChildren();
}
```

Related Variables

xbeeZB.remainingChildren → stores the number of remaining children available

9.5.3. Channel Verification

Specifies if Channel Verification is enabled or disabled (JV parameter). If JV=1, a router will verify the coordinator is on its operating channel when joining or coming up from a power cycle. If a coordinator is not detected, the router will leave its current channel and attempt to join a new PAN. If JV=0, the router will continue operating on its current channel even if a coordinator is not detected.

Example of use:

```
{
  xbeeZB.setChannelVerification(1);
  xbeeZB.getChannelVerification();
}
```

Related Variables

> `xbeeZB.channelVerification` → stores if Channel Verification is enabled or disabled

9.5.4. Join Notification

Specifies if a node should send a broadcast message when waking up or joining a network. This message is an identification message so as the other nodes can know if it is operating in the same network. It is recommended not to enable this parameter in big networks to prevent excessive broadcast messages.

Example of use:

```
{
  xbeeZB.setJoinNotification(1);
  xbeeZB.getJoinNotification();
}
```

Related Variables

> `xbeeZB.joinNotification` → stores if Join Notification is enabled or disabled

9.6. Node Discovery

Discovery nodes function is used to discover and to report all modules of its current operating channel and PAN ID.

9.6.1. Structure used in Discovery

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

```
struct Node
{
    uint8_t    MY[2];
    uint8_t    SH[4];
    uint8_t    SL[4];
    char       NI[20];
    uint8_t    PMY[2];
    uint8_t    DT;
    uint8_t    ST;
    uint8_t    PID[2];
    uint8_t    MID[2];
    uint8_t    RSSI;
};
```

- `MY`: 16-bit Network Address of the reported module.
- `SH[4]` and `SL[4]`: 64-bit MAC Source Address of the reported module.
- `NI`: Node Identifier of the reported module
- `PMY`: : Parent 16-bit network address. It specifies the 16-bit network address of its parent.
- `DT`: Device Type. It specifies if the node is a Coordinator, Router or End Device.
- `ST`: Status. Reserved by xbeeZB.
- `PID`: Profile ID. Profile ID used to application layer addressing.
- `MID`: Manufacturer ID. ID set by the manufacturer to identify the module.
- `RSSI`: Not returned in ZigBee.

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 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:

```
{
    xbeeZB.scanNetwork();
}
```

Related variables:

> `xbeeZB.totalScannedBrothers` → stores the number of discovered brothers
> `xbeeZB.scannedBrothers` → Node structure array that stores the info

- Scan network example:
  http://www.libelium.com/development/waspmote/examples/zb-10-scan-network

9.6.2. Searching nodes

Another possibility to discover a node is searching an 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 search. 16-bit and 64-bit addresses of the searched node will be stored in the packet send as an input.

Example of use

```
{
    uint8_t  mac[8];
    xbeeZB.nodeSearch("node01", mac);
}
```

Related variables

> `mac[0-7]` → Stores the 64-bit address of the searched node

• Search node example:
> http://www.libelium.com/development/waspmote/examples/zb-11a-node-search-tx
> http://www.libelium.com/development/waspmote/examples/zb-11b-node-search-rx

9.6.3. Node discovery to a specific node

When executing a Node Discovery all the nodes respond to it. If only one node is desired to respond, there is another possibility to find it. If its Node Identifier is known, a Node Discovery using its NI as an input can be executed.

Example of use

```
{
    xbeeZB.scanNetwork("node01");
}
```

Related variables

> `xbeeZB.totalScannedBrothers` → stores the number of discovered brothers. Must be '1'.

> `xbeeZB.scannedBrothers` → Node structure array that stores the info

9.6.4. Node Discovery Time

Is the amount of time a node will wait for responses from other nodes when performing a ND. Range: 0x20 - 0xFF [x 100 msec]. Default: 0x3C.

Example of use:

```
{
    uint8_t time[2]={0x3C,0x00};// In ZigBee is only used first array position
    xbeeZB.setScanningTime(time); // Set Scanning Time in ND
    xbeeZB.getScanningTime(); // Get Scanning Time in ND
}
```

Available Information

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

9.6.5. Node Discovery Options

Selects Node Discovery Options. Options available are:

• 0x01: Append DD value to ND responses.

• 0x02: Local device sends ND response frame when ND is issued.

Example of use:

```
{
    xbeeZB.setDiscoveryOptions(0x01); // Set Discovery Options for ND
    xbeeZB.getDiscoveryOptions(); // Get Discovery Options for ND
}
```

Available Information

xbeeZB.discoveryOptions → stores the selected options for ND

Note: *If the 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 1-2 seconds for setting the API function properly.*

When a Node Discovery is performed, due to it is a broadcast message, it will only reach as many hops as the set in 'Broadcast Max Hops' parameter. So, setting that parameter, a Node Discovery may only reach nodes near or may reach the entire network.

It is recommended to try on the final scenario to find the best value for this parameter.

10. Rebooting a Network

10.1. Network Reset

Resets network layer parameters on one or more modules within the PAN. Possible values are:

- 0: Resets network layer parameters on the node issuing the reset
- 1: Resets network layer parameters on the entire network. It sends a broadcast message to all the nodes to reset.

When a node resets itself, it leaves the PAN and starts the process of finding a PAN to join.

When a node resets the network, it sends a broadcast transmission to all the nodes within the PAN. When the Coordinator receives the message leaves the PAN and starts a new PAN. The rest of the nodes leave the PAN too, and start a process for joining a PAN. The time to reset the network depends on the hops the broadcast reset message needs to reach its destination. Our tests for a network with 3 hops discovered that time was around 3-4 seconds.

Example of use:

```
{
  xbeeZB.resetNetwork(0); //  Reset itself
  xbeeZB.resetNetwork(1); // Reset the entire network
}
```

• Coordinator resets network example:

http://www.libelium.com/development/waspmote/examples/zb-01b-coordinator-resets-network
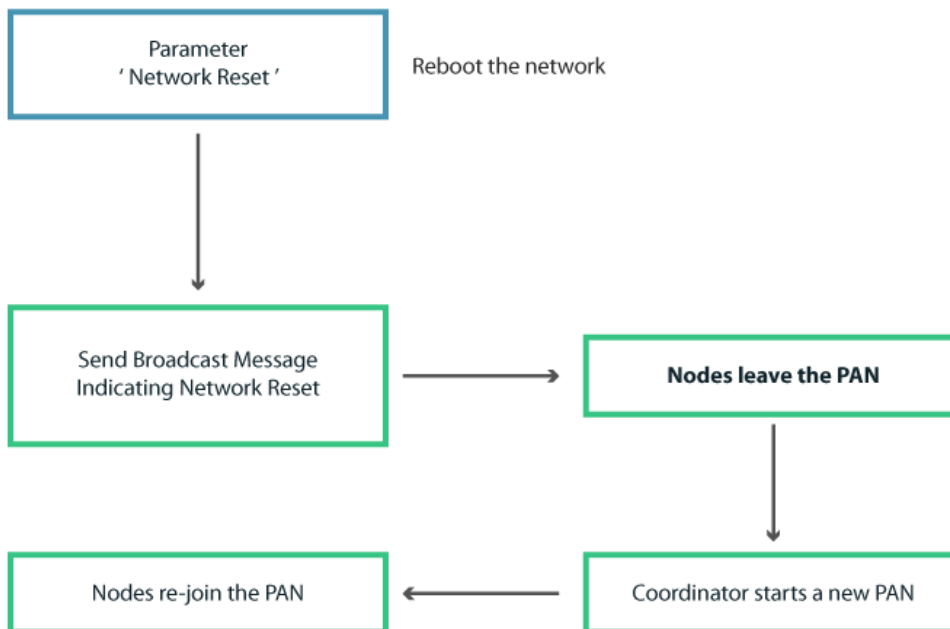

Figure : Network Reset Process

11. Sleep Options

11.1. Sleep Modes

Coordinators and Routers are not capable of sleeping, only End Devices can enter into a low-power state to safe battery and wake up when needed to transmit or receive data.

There are two sleep modes: pin sleep and cyclic sleep.

11.1.1. Pin Sleep

This mode is used to wake and sleep the module using a hardware pin. When the pin is asserted, the module enters to sleep after ending any transmission or reception in process. In the case the module has not joined a network, it will sleep after joining a network.

When a joined end device wakes up from a pin sleep, it sends a poll request to its parent to see if the parent has buffered data for the end device. The end device will continue to send poll requests every 100ms while it remains awake.

11.1.2. Cyclic Sleep

Cyclic sleep allows modules to wake up periodically to check for RF data and sleep when idle. In cyclic sleep mode, if serial or RF data is received, the module will start an inactivity timer and remain awake until this timer expires. While the module is awake, it will continue to send poll request transmissions to its parent to check for buffered data every 100ms. The timer will be restarted anytime serial or RF data is received. The module will resume sleep when the timer expires.

11.2. Sleep Parameters

There are some parameters involved in setting an End Device to sleep.

11.2.1. Sleep Mode

Sets the sleep mode for a module. Its available values are:

- 0: Sleep disabled. When using this mode, the module operates as a router.
- 1: Pin sleep enabled.
- 4: Cyclic Sleep enabled.
- 5: Cyclic Sleep with Pin awake enabled.

When 'Sleep Mode' changes to a non-zero value, the node leaves the PAN and rejoins it as an End Device.

Example of use:

```
{
  xbeeZB.setSleepMode(1);
  xbeeZB.getSleepMode();
}
```

Related Variables

xbeeZB.sleepMode → stores the sleep mode in a module

• Set sleep mode example:

http://www.libelium.com/development/waspmote/examples/zb-08-set-low-power-mode

11.2.2. Sleep Period

It determines how long a node will sleep continuously with a maximum of 28 seconds. The parent determines how long it will buffer a message for the sleeping device. It should be set at least equal to the longest sleeping value of any children.

Example of use:

```
{
  uint8_t asleep[3]={0xAF,0x00,0x00};
  xbeeZB.setSleepTime(asleep); //  Set Sleep period to 0,32 seconds
}
```

Related Variables

xbeeZB.sleepTime → stores the sleep period the module will be sleeping

11.2.3. Time Before Sleep

It determines the time a module will be awake waiting before sleeping. It resets each time data is received via RF or serial. Once the timer expires, the device will enter low-power state.

Example of use:

```
{
  uint8_t awake[3]={0x13,0x88,0x00};
  xbeeZB.setAwakeTime(awake);
}
```

Related Variables

xbeeZB.awakeTime → stores the time the module remains awake

11.2.4. Number of Sleep Periods

It determines the number of sleep periods to not assert the On/Sleep pin on wakeup if no RF data is waiting for the end device.

Example of use:

```
{
  uint8_t periods[2]={0x13,0x88};
  xbeeZB.setPeriodSleep(periods);
}
```

Related Variables

xbeeZB.periodSleep → stores the number of sleep periods

11.2.5. Sleep Options

It configures options for sleeping modes. Possible values are:

- 0x02 : Always awake for 'Time Before Sleep' time.
- 0x04 : Sleep entire 'Number of Sleep Periods' * 'Sleep Period' time.

Example of use:

```
{
  xbeeZB.setSleepOptions(0x02); //
}
```

Related Variables

xbeeZB.sleepOptions → stores the sleep option chosen

11.3. ON/OFF Mode

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).

Example of use:

```
{
  xbeeZB.setMode(XBEE_ON);
  xbeeZB.setMode(XBEE_OFF);
}
```

12. Synchronizing the Network

12.1. End Device Operation

End Devices rely on a parent (router or coordinator) to remain awake and receive any data packets addressed to them. When the end device wakes up from sleep, it sends a transmission (poll request) to its parent asking if the parent has received any RF data addressed to the end device. The parent, upon receipt of the poll request, will send an RF response and the buffered data (if present). If the parent has no data for the end device, the end device may return to sleep, depending on its sleep mode configuration settings.

If the end device is awake, it will send poll requests every 100ms to ensure it receives any new RF data from its parent. When RF data is sent to a sleeping end device, its parent buffers the data until the end device polls for the data or a timeout occurs.
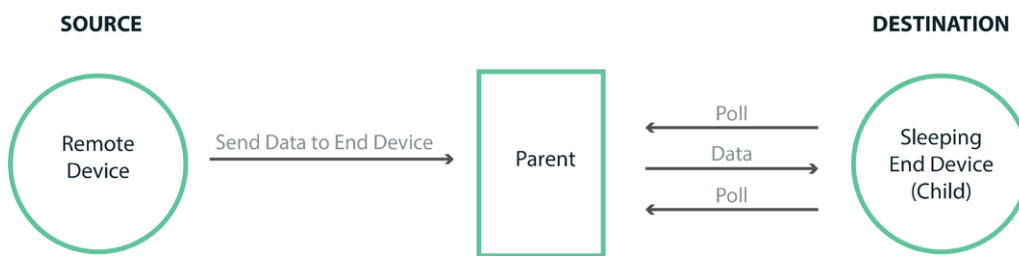

Figure : RF Data Sending to Sleeping End Device

12.2. Parent Operation

When an end device joins a network, it becomes a child of the device it joined to, and the device that allowed the join becomes the end device's parent. Thereafter, the parent will manage RF data packets for the end device. The actual storage time is computed as ('Sleep Period * 2.5), not exceeding 30 seconds. If end devices implement cyclic sleep, 'Sleep Period' should be set the same on a parent as it is on their sleeping end device children. In the case of pin sleep, where RF data could be received, the end device should wake within 'Sleep Period' time to ensure incoming RF data is not lost. The parent can only store one broadcast packet (the most recently received) for its end device children. If the parent receives an RF packet addressed to the end device, it will store the data packet until one of the following events occurs:

• The parent runs out of storage space and cannot store a new packet.
• A packet has been stored for a long period of time (timeout).
• The destination end device child sends a poll request transmission to request the data packet.

When the parent stores a packet addressed to an end device child, it stores the packet for a maximum time set by 'Sleep Period' parameter.

The parent is also responsible for performing any route or address discoveries to forward data sent by its end device children into the mesh network.
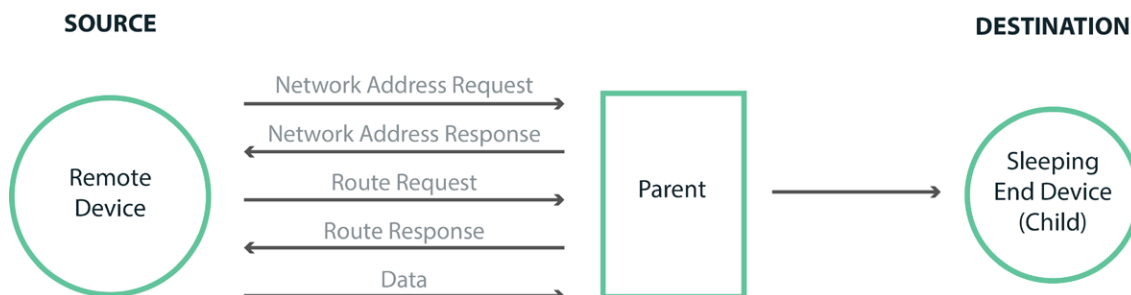

Figure : Determining Sleeping End Device Address and Route

12.3. Cyclic Sleep Operation

In cyclic sleep, the time the module sleeps depends on several configuration parameters as 'Sleep Period', 'Number of Sleep Periods' or 'Sleep Options'.

A parent can only buffer data for30 seconds and an end device can only sleep a maximum of 28 seconds. When it wakes up, it sends a poll request to check if there is data available. This working mode ensures data is not lost even if an end device is slept the maximum time.

In many cases, the On/Sleep pin can be used to wake up an external microprocessor or peripheral device when the module wakes up from sleep. If the end device wakes up and finds that its parent has no data, there will be no need to wake up the external device. The 'Number of Sleep Periods' parameter is a multiplier of 'Sleep Period' parameter that determines how often to set the On/Sleep pin when waking.

In some applications, the end device may transmit data at a very slow rate (once an hour, once a day, etc) and will only receive data in response to its transmission. In such cases, the 'Sleep Options' parameter can be used to cause an end device sleeping for the entire 'Sleep Period' * 'Number of Sleep Periods' time.

Since a parent can only buffer data for 30 seconds, an end device should not sleep more than 30 seconds unless it does not have to receive any data.

13. Security and Data Encryption

13.1. ZigBee Security and Data encryption Overview

ZigBee security and data encryption is based on security defined in 802.15.4 protocol. The encryption algorithm used in ZigBee 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 that 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.
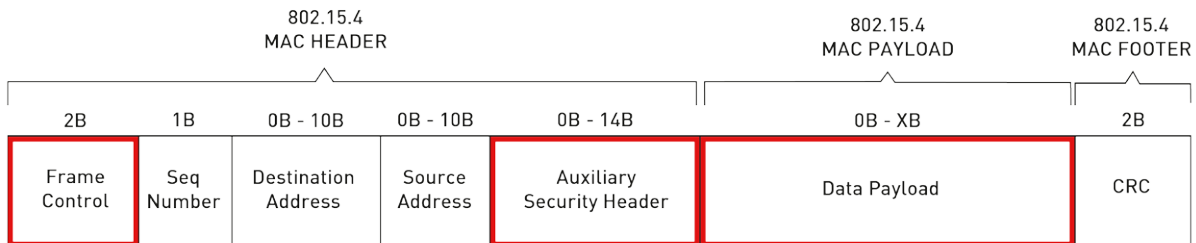

Figure : IEEE 802.15.4 Frame

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.
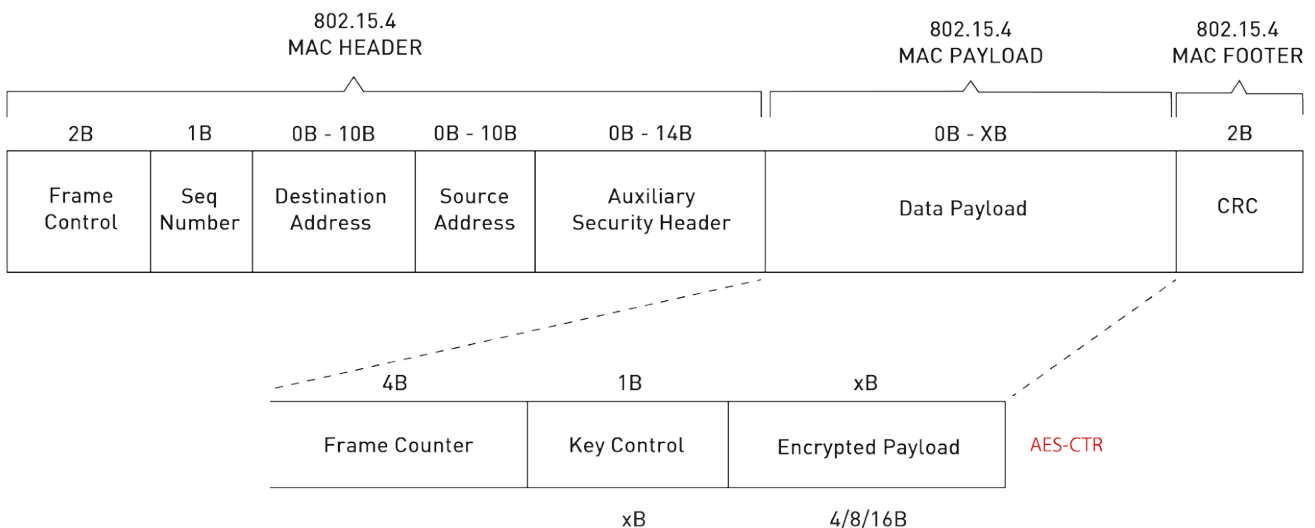

Figure : AES-CTR Encryption Frame

ZigBee implements **two extra security** layers on top of the 802.15.4 one: the **Network** and **Application** security layers. All the security policies rely on the AES 128b encryption algorithm so the hardware architecture previously deployed for the link level (MAC layer) is still valid. There are three kind of keys: master, link and network.

- **Master Keys**: They are pre-installed in each node. Their function is to keep confidential the Link Keys exchange between two nodes in the **Key Establishment Procedure** (**SKKE**).
- **Link Keys**: They are unique between each pair of nodes. These keys are managed by the Application level. The are used to encrypt all the information between each two devices, for this reason more memory resources are needed in each device.
- **Network key**: It is a unique 128b key shared among all the devices in the network. It is generated by the **Trust Center** and regenerated at different intervals. Each node has to get the Network Key in order to join the network. Once the trust center decides to change the Network Key, the new one is spread through the network using the **old Network Key**. Once this new key is updated in a device, its **Frame Counter** is initialized to zero. This **Trust Center** is normally the Coordinator, however, it can be a dedicated device. It has to **authenticate and validate** each device which attempts to join the network.

13.2. Security in API libraries

As explained previously, ZigBee provides secure communications inside a network using 128bits AES encryption. The API functions enable using security and data encryption.

13.2.1. Encryption Enable

Enables the 128bits AES encryption in the modules.

Example of use:

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

Related Variables

   `xbeeZB.encryptMode` → stores if security is enabled or not

• Enable encryption example:

   http://www.libelium.com/development/waspmote/examples/zb-12a-setting-encryption-mode-coordinator

   http://www.libelium.com/development/waspmote/examples/zb-12b-setting-encryption-mode-routers

13.2.2. Encryption Options

Configures options for encryption. Available values are:

- 0x01: Send the security key unsecured over the air during joins
- 0x02: Use Trust Center. ZigBee defines a trust center device that is responsible for authenticating devices that join the network. The trust center also manages link key distribution in the network.

Example of use:

```
{
    xbeeZB.setEncryptionOptions(0x02); //  Use Trust Center
    xbeeZB.getEncryptionOptions(); // Get options for encryptions
}
```

Related Variables

   `xbeeZB.encryptOptions` → stores the options for security

13.2.3. Encryption Key

Sets the 128bits AES Encryption Key. If this key is set to zero, a random key is chosen.

Example of use

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

Related Variables

>   xbeeZB.linkKey → stores the key that has been set in the network

13.2.4. Link Key

Sets the 128bits AES Link Key. If 'Link Key' is set to zero, the coordinator will send the network key in clear to joining devices.

Example of use

```
{
  char* KEY="WaspmoteNetwKey!";
  xbeeZB.setNetworkKey(KEY); // Set Link Key
}
```

Related Variables

>   xbeeZB.networkKey → stores the key that has been set in the network

13.2.5. Application Level Security

Set the application level security. This function decreases the payload by 4 bytes.

Example of use

```
{
  xbeeZB.setAPSencryption(XBEE_ON); // Enables APS encryption
  xbeeZB.setAPSencryption(XBEE_OFF); // Disables APS encryption
}
```

13.3. Security in the network

If encryption is enabled in the coordinator, it will apply a security policy to the network when creating it. Enabling security in a network adds an authentication step to the joining process. For example, after a router joins a network, it must then obtain the network security key to become authenticated. If the device cannot obtain the network security key, authentication fails, and the device leaves the network since it cannot communicate with anyone in the network.

13.3.1. Trust Center

A trust center is a single device that is responsible for determining who may join the network. If a trust center is enabled, the trust center must approve all devices before joining the network. If a router allows a new device to join the network, the router sends a notification to the trust center that a join has occurred. The trust center instructs the router to either authenticate the newly joined device or to deny it. A trust center is required for some public ZigBee profiles.

To use a trust center in a ZigBee network, the coordinator should set the "use trust center" bit correctly in the 'Encryption Options' parameter before starting a network. Only the coordinator can serve as a Trust Center.

13.3.2. Managing Security Keys

Modules define a network key and a link key (trust center link key). Both keys are 128-bits and are used to apply AES encryption to RF packets.

The coordinator selects a network security key using the 'Encryption Key' parameter. Similarly, the coordinator must also specify a link key using the 'Link Key' parameter.

When a device joins a secure network, it must obtain the network key from the coordinator. The coordinator will either transmit the network key using a pre-installed link key. If the 'Encryption Options' bit is set to transmit the network key unencrypted, or if the 'Link Key' parameter is set to 0 on the coordinator (select a random link key), the coordinator will transmit the network key unencrypted. Otherwise, if the 'Encryption Options' bit is not set and 'Link Key' is > 0, the coordinator will encrypt the network key with the link key.

If a joining device does not have the right preconfigured link key, and the network key is being sent encrypted, the joining device will not be able to join the network.

Network security requires a 32bits frame counter maintained by each device. This frame counter is incremented after each transmission and cannot wrap to 0. If a neighbour receives a transmission with a frame counter that is less than or equal to the last received frame counter, the packet will be discarded.

To prevent an eventual lockup where the frame counter on a device reaches 0xFFFFFFFF, the network key should be periodically updated (changed) on all devices in the network. To update the network key in the network, the coordinator should issue the 'Encryption Key' parameter with a new security key. This will send a broadcast transmission throughout the network causing the frame counters on all devices to reset to 0, and causing devices to begin using the new network key. All devices will also retain the previous key for a short time until everyone has switched to the new key.

14. Code examples and extended information

In the Waspmote Development section you can find complete examples:

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

Example:

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

// Destination MAC address
//////////////////////////////////////
char RX_ADDRESS[] = "0013A200403A2A49";
//////////////////////////////////////

// Define the Waspmote ID
char WASPMOTE_ID[] = "node_01";

// define variable
uint8_t error;


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

  // store Waspmote identifier in EEPROM memory
  frame.setID( WASPMOTE_ID );

  // init XBee
  xbeeZB.ON();

  delay(3000);

  //////////////////////////
  // 2. check XBee's network parameters
  //////////////////////////
  checkNetworkParams();
}


void loop()
{
  //////////////////////////////////////
  // 1. Create ASCII frame
  //////////////////////////////////////

  // create new frame
  frame.createFrame(ASCII);

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


  //////////////////////////////////////
  // 2. Send packet
  //////////////////////////////////////

  // send XBee packet
  error = xbeeZB.send( RX_ADDRESS, frame.buffer, frame.length );
```

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

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



/*******************************************
 *
 *   checkNetworkParams - Check operating
 *   network parameters in the XBee module
 *
 *******************************************/
void checkNetworkParams()
{
  // 1. get operating 64-b PAN ID
  xbeeZB.getOperating64PAN();

  // 2. wait for association indication
  xbeeZB.getAssociationIndication();

  while( xbeeZB.associationIndication != 0 )
  {
    delay(2000);

    // get operating 64-b PAN ID
    xbeeZB.getOperating64PAN();

    USB.print(F("operating 64-b PAN ID: "));
    USB.printHex(xbeeZB.operating64PAN[0]);
    USB.printHex(xbeeZB.operating64PAN[1]);
    USB.printHex(xbeeZB.operating64PAN[2]);
    USB.printHex(xbeeZB.operating64PAN[3]);
    USB.printHex(xbeeZB.operating64PAN[4]);
    USB.printHex(xbeeZB.operating64PAN[5]);
    USB.printHex(xbeeZB.operating64PAN[6]);
    USB.printHex(xbeeZB.operating64PAN[7]);
    USB.println();

    xbeeZB.getAssociationIndication();
  }

  USB.println(F("\nJoined a network!"));

  // 3. get network parameters
  xbeeZB.getOperating16PAN();
  xbeeZB.getOperating64PAN();
  xbeeZB.getChannel();

  USB.print(F("operating 16-b PAN ID: "));
  USB.printHex(xbeeZB.operating16PAN[0]);
  USB.printHex(xbeeZB.operating16PAN[1]);
  USB.println();

  USB.print(F("operating 64-b PAN ID: "));
  USB.printHex(xbeeZB.operating64PAN[0]);
  USB.printHex(xbeeZB.operating64PAN[1]);
```

```
    USB.printHex(xbeeZB.operating64PAN[2]);
    USB.printHex(xbeeZB.operating64PAN[3]);
    USB.printHex(xbeeZB.operating64PAN[4]);
    USB.printHex(xbeeZB.operating64PAN[5]);
    USB.printHex(xbeeZB.operating64PAN[6]);
    USB.printHex(xbeeZB.operating64PAN[7]);
    USB.println();

    USB.print(F("channel: "));
    USB.printHex(xbeeZB.channel);
    USB.println();
}
```

15. API changelog

Keep track of the software changes on this link:

www.libelium.com/development/waspmote/documentation/changelog/#ZigBee

16. Documentation changelog

From v4.7 to v4.8

- References to the new LoRaWAN module

From v4.6 to v4.7

- References to the new Sigfox module

From v4.5 to v4.6

- Changes in sending/receiving chapters. New function prototypes are used

From v4.4 to v4.5:

- References to the new LoRa module
- Link to the new online API changelog

From v4.3 to v4.4:

- Expansion Radio Board section updated
- API changelog updated to API v011

From v4.2 to v4.3:

- References to XBee ZigBee 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.