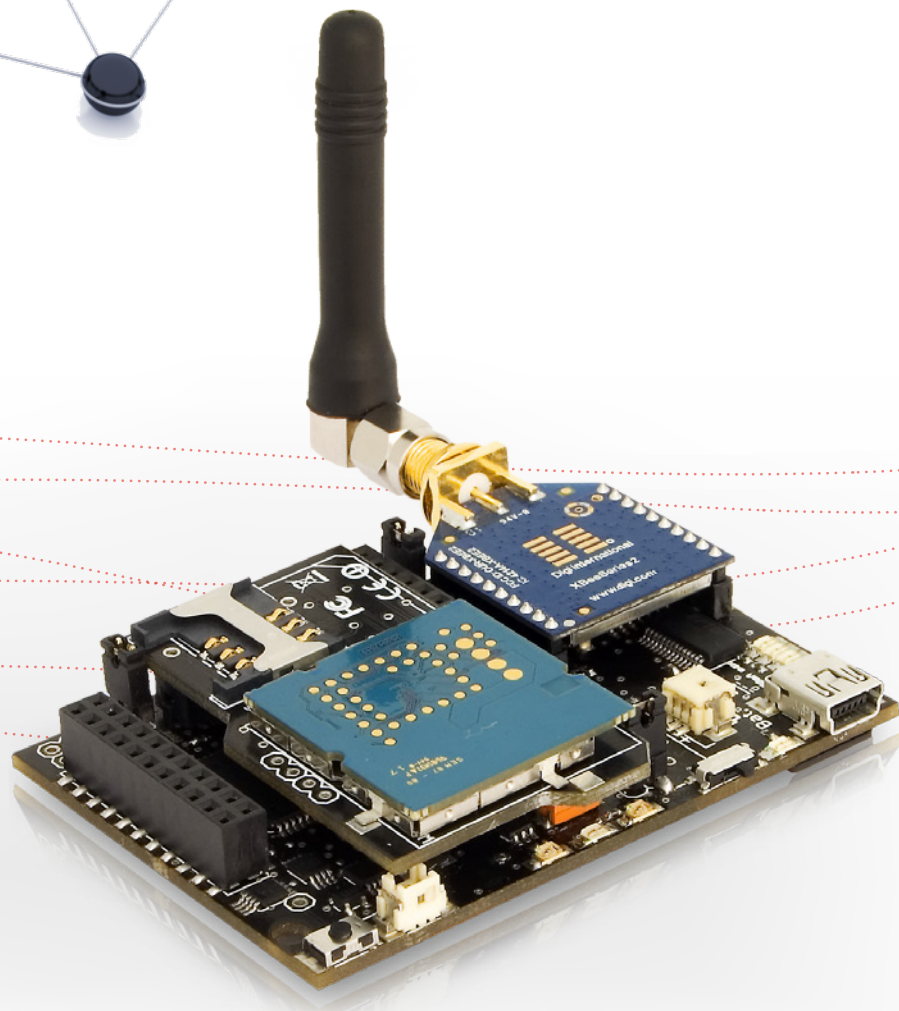


Wasp mote ZigBee

Networking Guide



Document Version: v1.0 - 11/2012
© Libelium Comunicaciones Distribuidas S.L.

INDEX

1. Hardware	6
2. General Considerations.....	8
2.1. Waspote Libraries.....	8
2.1.1. Waspote XBee Files.....	8
2.1.2. Constructor.....	8
2.2. API Functions.....	8
2.3. API extension.....	9
2.4. Waspote reboot	9
2.5. Constants pre-defined	9
3. Initialization.....	10
3.1. Initializing	10
3.2. Setting ON	12
3.3. Setting OFF.....	12
4. Node Parameters	13
4.1. MAC Address.....	13
4.2. Network Address	13
4.3. Extended PAN ID	13
4.4. Operating Extended PAN ID.....	14
4.5. Operating PAN ID	14
4.6. Node Identifier	14
4.7. Channel.....	15
4.8. Device Type Identifier	16
4.9. 16-bit Address Importance	16
4.10. Node Types	16
5. Packet Parameters	18
5.1. Structure used in packets.....	18
5.2. Maximum payloads.....	21
5.2.1. Maximum Payload	21
6. Power Gain and Sensibility	22
6.1. Power Level	22
6.2. Power Mode.....	23
6.3. Received Signal Strength Indicator	23
7. Radio Channels	24
7.1. Scan Channels	24
7.2. Scan Duration.....	24

8. Connectivity	25
8.1. Topologies	25
8.2. ZigBee Architecture	26
8.2.1. ZigBee Device Objects	26
8.2.2. ZDO Management Plane	26
8.2.3. Application Support Sub-layer (APS)	26
8.2.4. Network Layer	26
8.2.5. Security Plane	27
8.2.6. Application Framework	27
8.2.7. Service Access Points	27
8.2.8. Application Endpoints	27
8.2.9. Application Profiles	27
8.2.10. Attributes	27
8.2.11. Clusters	28
8.2.12. Binding	28
8.2.13. Wasp mote Application Level	28
8.3. ZigBee Addressing	29
8.3.1. Device Addressing	29
8.3.2. Application Layer Addressing	29
8.4. Connections	29
8.4.1. Broadcast	29
8.4.2. Unicast	29
8.4.3. Many to one	30
8.4.4. Parameters involved	30
8.5. Discovering and Searching Nodes	31
8.5.1. Structure used in Discovery	31
8.5.2. Searching nodes	32
8.5.3. Node discovery to a specific node	33
8.6. Sending Data	33
8.6.1. API Frame Structure	33
8.6.2. Application Header	33
8.6.3. Fragmentation	34
8.6.4. Structure packetXBee	34
8.6.5. Sending without API header	34
8.6.6. Examples	34
8.7. Receiving Data	38
8.7.1. Examples	39
9. Starting a Network	42
9.1. Coordinator Startup	42
9.2. Coordinator wake up process from Reset	42
9.3. Related Parameters	43
9.3.1. Node Join Time	43
9.3.2. ZigBee Stack Profile	43

9.4. Examples	43
9.4.1. Coordinator Startup Process. Random Extended PAN ID. All channels to scan.....	43
9.4.2. Coordinator Startup Process. Selected Extended PAN ID. Only some channels to scan	44
10. Joining an Existing Network.....	45
10.1. Router Joining a Network.....	45
10.1.1. Example.....	45
10.2. End Device Joining a Network.....	45
10.2.1. Example.....	46
10.3. Router / End Device Wake Up from Reset Process.....	47
10.4. Channel Verification	47
10.4.1. Router Channel Verification	47
10.4.2. End Device Channel Verification	47
10.5. Related Parameters.....	47
10.5.1. Association Indication.....	47
10.5.2. Number of Remaining Children	48
10.5.3. Channel Verification.....	48
10.5.4. Join Notification	48
10.6. Node Discovery	49
10.6.1. Node Discovery	49
10.6.2. Node Discovery Time.....	49
10.6.3. Node Discovery Options	50
11. Rebooting a Network	51
11.1. Network Reset.....	51
12. Sleep Options.....	52
12.1. Sleep Modes	52
12.1.1. Pin Sleep.....	52
12.1.2. Cyclic Sleep	52
12.2. Sleep Parameters	52
12.2.1. Sleep Mode	52
12.2.2. Sleep Period.....	52
12.2.3. Time Before Sleep	53
12.2.4. Number of Sleep Periods	53
12.2.5. Sleep Options.....	53
12.3. ON/OFF Mode.....	54
13. Synchronizing the Network	55
13.1. End Device Operation.....	55
13.2. Parent Operation	55
13.3. Cyclic Sleep Operation.....	56
14. Security and Data Encryption	57
14.1. ZigBee Security and Data encryption Overview	57

14.2. Security in API libraries	58
14.2.1. Encryption Enable	58
14.2.2. Encryption Options	58
14.2.3. Encryption Key	58
14.2.4. Link Key	58
14.2.5. Application Level Security	59
14.3. Security in the network	59
14.3.1. Trust Center	59
14.3.2. Managing Security Keys	59
14.4. Examples	60
14.4.1. Configuring a network using security without a Trust Center	60
14.4.2. Configuring a network using security and the coordinator as a Trust Center	60
15. Code examples and extended information	61

1. Hardware

Module	Frequency	Transmission Power	Sensitivity	Number of channels	Distance
XBee-ZB	2,40 – 2,48GHz	2mW	-96dBm	16	500m
XBee-ZB-PRO	2,40 – 2,48GHz	50mW	-102dBm	13	7000m

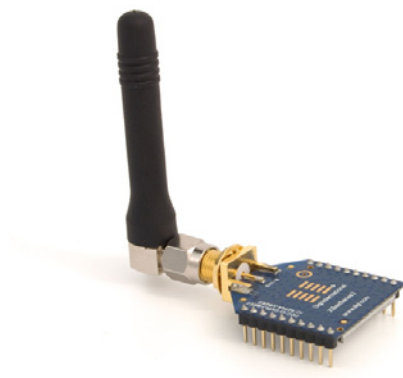


Figure 1: XBee ZigBee



Figure 2: XBee ZigBee PRO

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 13.

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.

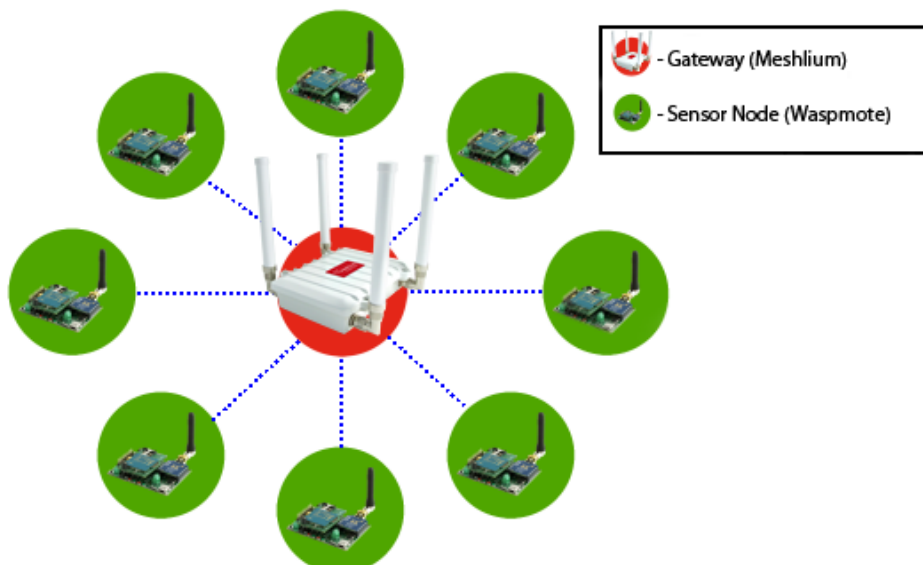


Figure 3: Star topology

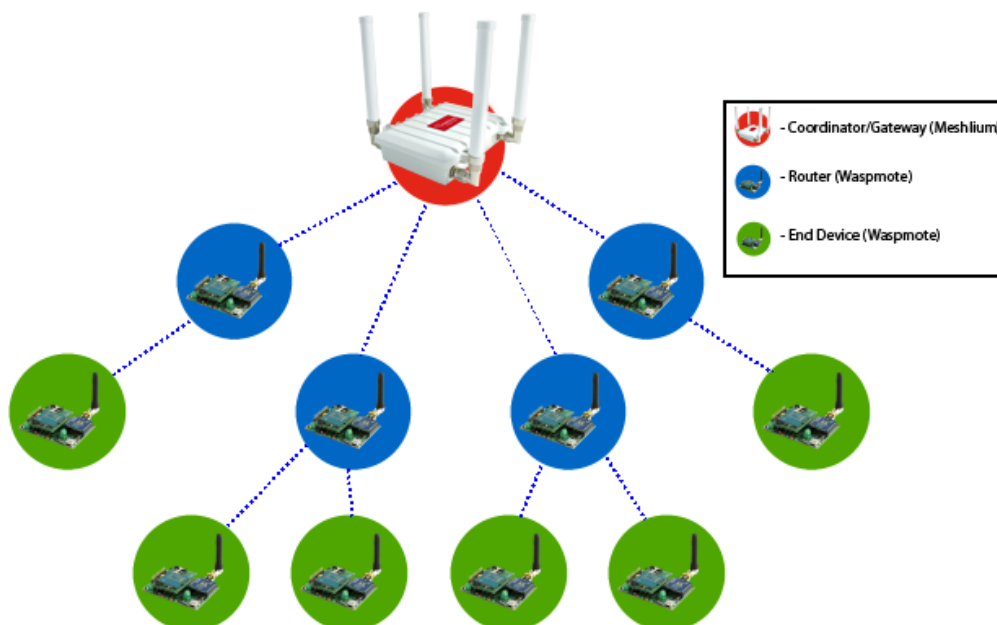


Figure 4: Tree topology

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

Parameter	Tx XBee ZB
0	-8dBm
1	-4dBm
2	-2dBm
3	0dBm
4	2dBm

Figure 5: Transmission power values

A mode called **boost** can be set up which improves reception sensibility by **1dB** and transmission power by **2dB**. Using this mode improves reach, but also increases consumption.

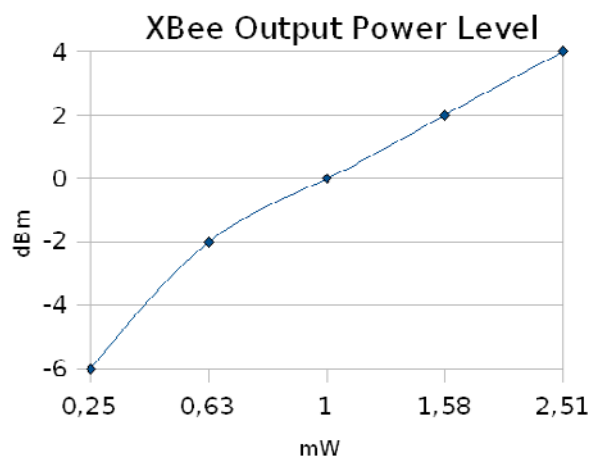


Figure 6: XBee Output Power Level (Boost Mode ON)

2. General Considerations

2.1. Wasmote Libraries

2.1.1. Wasmote XBee Files

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

2.1.2. Constructor

To start using Wasmote XBee library, an object from class 'WaspXBeeZB' must be created. This object, called 'xbeeZB', is created inside Wasmote XBee library and it is public to all libraries. It is used through the guide to show how the Wasmote 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[0-100] → stores the response given by the module up to 100 bytes

2.4. Waspmote reboot

When Waspmote is rebooted or it wakes up from a deep sleep state (battery is disconnected) 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 'WaspXBeeConstants.h'. These constants define some parameters like the size of each fragment or maximum data size. The most important constants are explained next:

- **MAX_DATA**: 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.
- **DATA_MATRIX**: it defines the data size for each fragment. As the maximum payload is 100bytes, there is no reason to make it bigger.
- **MAX_PARSE**: it defines the maximum data that is received 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**: it defines the maximum number of brothers that can be stored.
- **MAX_FRAG_PACKETS**: it defines the maximum number of fragments that can be received from a global packet. If a packet is divided in more fragments, when it is detected all the fragments will be deleted and the packet will be discarded.
- **MAX_FINISH_PACKETS**: it defines the maximum number of finished packets that can be stored.
- **DATA_OFFSET**: it is used as an input parameter in 'setDestinationParams'. It specifies that the data given as a parameter must be added at the end of the packet. It allows the generation of one packet from several calls to 'setDestinationParams' with different types of data.
- **DATA_ABSOLUTE**: it is used as an input parameter in 'setDestinationParams'. It specifies that the data given as a parameter are the data to send. It also sets the addresses to the packet.
- **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. Initializing

It initializes all the global variables that will be used later.

It returns nothing.

The initialized variables are:

- **protocol** : specifies the protocol used (802.15.4 in this case).
- **freq** : specifies the frequency used (2,4GHz in this case).
- **model** : specifies the model used. There are two possible models: Normal(2mW) or PRO(50mW).
- **totalFragmentsReceived** : specifies the number of fragments expected from a packet
- **pendingPackets** : specifies the packets pending of fragments to be completed
- **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 an 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.init(ZIGBEE,FREQ2_4G,NORMAL); // initializes the variables
}
```

Expansion Radio Board (XBee ZigBee)

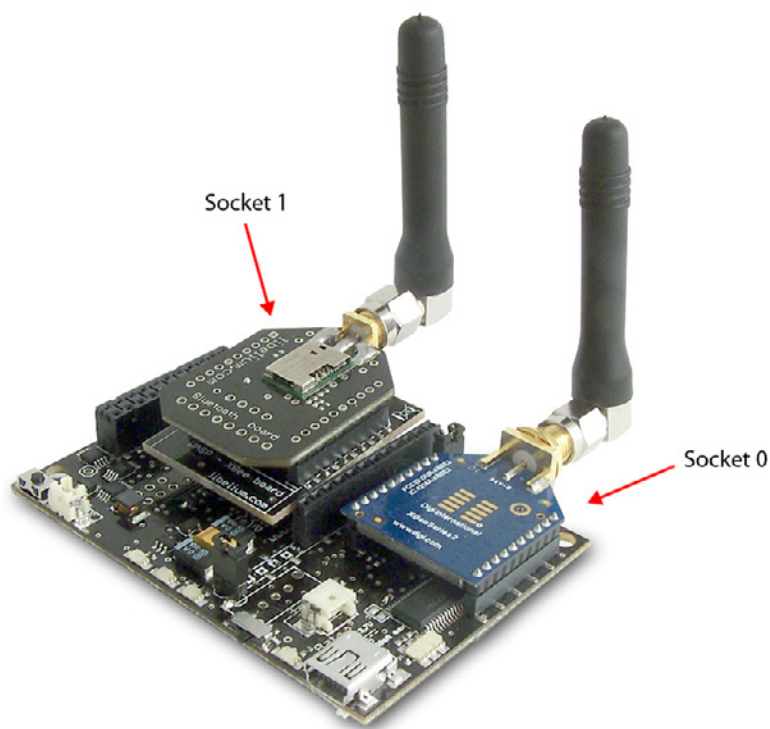
The new Expansion Board allows to connect two radios at the same time in the Wasp mote sensor platform. This means a lot of different combinations are now possible using any of the six radios available for Wasp mote: 802.15.4, ZigBee, Bluetooth, RFID, Wifi, GPRS, 868 MHz and 900 MHz.

Some of the possible combinations are:

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

Remark: the GPRS module does not need the Expansion Board to be connected to Wasp mote. 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 in order to initialize the XBee module called 'init'. This function supports a new parameter which permits to select the UART. It is possible to choose between UART0 or UART1.

The initialization function is the following:

```
xbeeZB.init(ZIGBEE,FREQ2_4G,NORMAL,UART);
```

Selecting UART0:

```
xbeeZB.init(ZIGBEE,FREQ2_4G,NORMAL,UART0);
```

Selecting UART1:

```
xbeeZB.init(ZIGBEE,FREQ2_4G,NORMAL,UART1);
```

In 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 UART0 regular socket.

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

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

Finally, it is necessary to initialize both modules. For example, xbeeZB is initialized in UART0 and xbeeZB_2 in UART1 as follows:

```
xbeeZB.init(ZIGBEE,FREQ2_4G,NORMAL,UART0);
```

```
xbeeZB_2.init(ZIGBEE,FREQ2_4G,NORMAL,UART1);
```

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 NO₂/O₃ 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 ON

It opens the UART and switches the XBee ON. The baud rate used to open the UART is defined on the library (38400bps by default).

Example of use

```
{  
  xbeeZB.ON(); // Opens the UART and switches the XBee ON  
}
```

3.3. Setting OFF

It closes the UART and switches the XBee OFF.

Example of use

```
{  
  xbeeZB.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 later in the network, and some parameters necessary for using the API functions.

4.1. 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

4.2. 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 address 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(); // Get Network Address
}
```

Related Variables

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

4.3. 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.

The Coordinator selects a random extended PAN ID when the network is started the extended PAN ID is set in the module. When a node joins a network, during the joining process.

If Extended PAN ID is set to '0' before starting a network, the Coordinator will select a random PAN ID, but if a Extended PAN ID is set before starting a network, this PAN ID will be set.

If a joining node has set its Extended PAN ID, it will only join a network with that PAN ID.

Example of use

```
{
  panid={1,2,3,4,5,6,7,8}; // array containing the PAN ID
  xbeeZB.setPAN(panid); // Set PANID
  xbeeZB.getPAN(); // Get PAN ID
}
```

Related Variables

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

NOTE: 'Extended PAN ID' is used when a specific ID wants to be used, setting this parameter to a value different from zero will make the module only chooses this ID. If it is set to zero, any ID will be chosen.

4.4. Operating Extended PAN ID

64-bit number that specifies the Operating Extended PAN ID used in the network. If Extended PAN ID is set to '0', Operating Extended PAN ID will be set to the randomly selected PAN ID (see below). If Extended PAN ID is set to a value different of '0', Operating Extended PAN ID will be set to that value.

Example of use:

```
{
  xbeeZB.getExtendedPAN(); // Get Operating Extended PAN ID
}
```

Related Variables

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

NOTE: 'Operating Extended PAN ID' is the current ID the module is working on. If 'Extended PAN ID' is set to a value different from zero, both will have the same value. If 'Extended PAN ID' is set to zero, the module will select randomly a value to use as PAN ID.

4.5. Operating PAN ID

16-bit number that specifies the Operating PAN ID used in the network. It reflects the 16-bit Operating PAN ID the module is running on.

Example of use

```
{
  xbeeZB.getOperatingPAN(); // Get Operating PAN ID
}
```

Related Variables

xbeeZB.operatingPAN[0-1] → stores the 16-bit Operating PAN ID

NOTE: 'Operating PAN ID' is the current ID the module is working on, but specified only with 16 bits instead of using 64 bits. It is used to reduce overheads on application level tasks.

4.6. Node Identifier

A max 20-character ASCII string 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("forrestnode-01#"); // Set 'forrestnode-01' as NI
  xbeeZB.getNodeIdentifier(); // Get NI
}
```

Related Variables

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

4.7. 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.

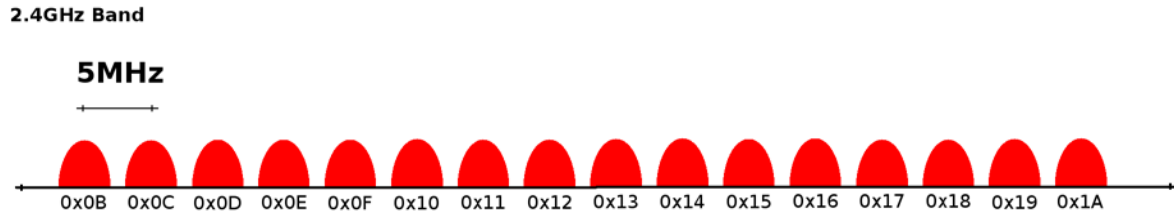


Figure 7: Operating Frequency Bands

Channel Number	Frequency	Supported by
0x0B – Channel 11	2,400 – 2,405 GHz	Normal / PRO
0x0C – Channel 12	2,405 – 2,410 GHz	Normal / PRO
0x0D – Channel 13	2,410 – 2,415 GHz	Normal / PRO
0x0E – Channel 14	2,415 – 2,420 GHz	Normal / PRO
0x0F – Channel 15	2,420 – 2,425 GHz	Normal / PRO
0x10 – Channel 16	2,425 – 2,430 GHz	Normal / PRO
0x11 – Channel 17	2,430 – 2,435 GHz	Normal / PRO
0x12 – Channel 18	2,435 – 2,440 GHz	Normal / PRO
0x13 – Channel 19	2,440 – 2,445 GHz	Normal / PRO
0x14 – Channel 20	2,445 – 2,450 GHz	Normal / PRO
0x15 – Channel 21	2,450 – 2,455 GHz	Normal / PRO
0x16 – Channel 22	2,455 – 2,460 GHz	Normal / PRO
0x17 – Channel 23	2,460 – 2,465 GHz	Normal / PRO
0x18 – Channel 24	2,465 – 2,470 GHz	Normal / PRO
0x19 – Channel 25	2,470 – 2,475 GHz	Normal
0x1A – Channel 26	2,475 – 2,480 GHz	Normal

Figure 8: Channel Frequency Numbers

Example of use:

```
{
  xbeeZB.getChannel(); // Get Channel
}
```

Related Variables

xbeeZB.channel → stores the operating channel

4.8. Device Type Identifier

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

Example of use

```
{
  uint8_t devicetype[4]={1,2,3,4}; // Device Type Identifier
  xbeeZB.setDeviceType(devicetype); // Set Device Type Identifier
  xbeeZB.getDeviceType(); // Get Device Type Identifier
}
```

Related Variables

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

4.9. 16-bit Address Importance

ZigBee networks are called personal area networks or PANs. Each network is defined with a unique PAN identifier. Modules support both 64-bit (extended) PAN ID and 16-bit PAN ID. The 16-bit PAN ID is used in all data transmissions, while, the 64-bit PAN ID is used during joining, and to resolve 16-bit PAN ID conflicts that may occur.

Modules can be identified by their unique 64-bit addresses or a user-configurable ASCII string identifier. ZigBee uses 16-bit addresses to reduce overhead in packets, so 16-bit addresses are needed to send packets inside a ZigBee network.

If the 16-bit address of a module is known, it is specified in the packet and the transmission will start once the route has been discovered. If the 16-bit address of a module is not known, a process called 'Address Discovery' will be executed to discover its 16-bit address.

ZigBee addressing is explained in chapter 8 with more detail.

4.10. Node Types

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

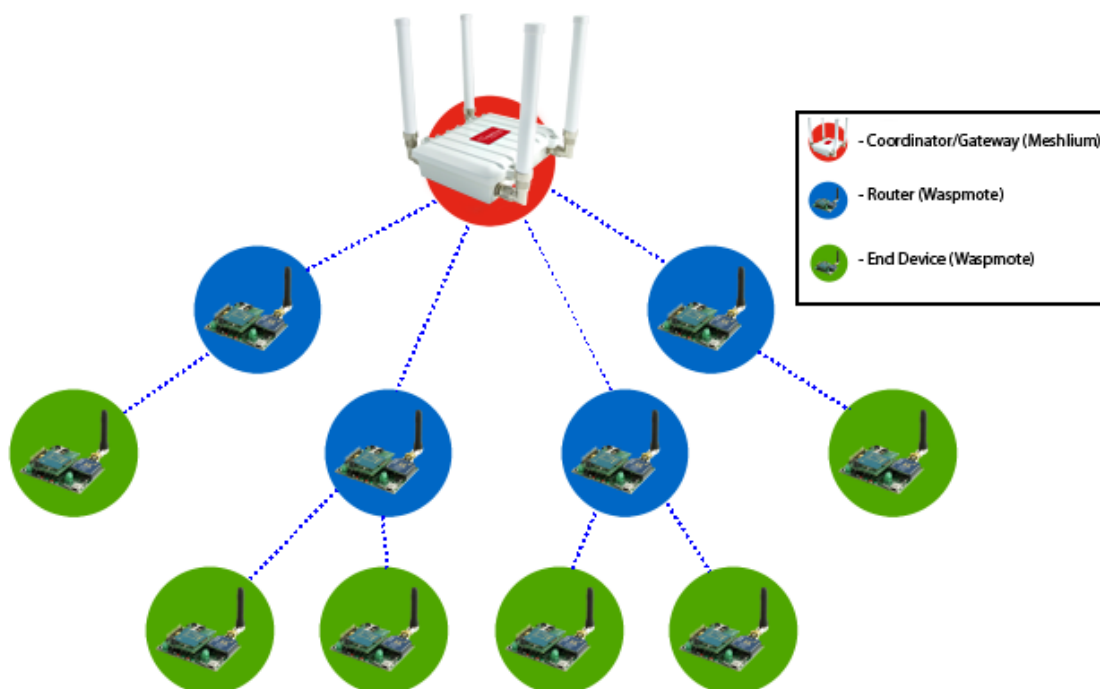


Figure 9: Tree topology

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.

5. Packet Parameters

5.1. Structure used in packets

Packets are structured in API libraries 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 ; 2=cluster ; 3=synchronization
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]
uint16_t frag_length; // Fragment length. Used to send each fragment of a big packet
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;         // options: 0x08=Multicast transmission

/***** APPLICATION *****/
uint8_t packetID;    // ID for the packet
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 macOL[4];    // 32b Lower Mac Origin Source
uint8_t macOH[4];    // 32b Higher Mac Origin Source
uint8_t naO[2];      // 16b Network Address origin
char niO[20];        // Node Identifier Origin. To use in transmission, it must finish "#".
uint8_t RSSI;        // Receive Signal Strength Indicator
uint8_t address_typeS; // 0=16B ; 1=64B
uint8_t typeSourceID; // 0=naS ; 1=macSource ; 2=NI
uint8_t numFragment; // Number of fragment to order the global packet
uint8_t endFragment; // Specifies if this fragment is the last fragment of a global packet
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

```

- **macDL & macDH**
64-bit destination address. It is used to specify the MAC address of the destination node.
- **mode**
Transmission mode chosen to transmit that packet. Available values :
 - 0 : Unicast transmission
 - 1 : Broadcast transmission
 - 2 : Application binding transmission, using clusters and endpoints
 - 3 : Synchronization packet

- **address_type**
Address type chosen, between 16-bit and 64-bit addresses. Always set to '1' in ZigBee.
 - 0 : 16-bit address
 - 1 : 64-bit address
- **naD**
16-bit Destination Network Address. It is used in data transmissions.
- **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. The API function responsible for sending data will take care of fragmenting the packet if it exceeds the maximum payload.

Its max 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.
- **frag_length**
Fragment data length. It is used by the API function responsible for sending data in the internal process of fragmenting packets.
- **SD**
Source Endpoint.
- **DE**
Destination Endpoint.
- **CID**
Cluster ID.
- **PID**
Profile ID.
- **MY_known**
It specifies if the 16-bit address is known before sending the packet. Available values are:
 - 0 : 16-bit address unknown
 - 1 : 16-bit address known
- **opt**
Options for the transmitted packet. Options available in ZigBee are:

0x08: Enables Multicast transmission.
- **packetID**
ID used in the application level to identify the packet. It is filled by the transmitter and it is used by the receiver to manage the received packet and the pending fragments.
- **macSL & macSH**
64-bit Source MAC Address. It is filled by the receiver, and it specifies the address of the node which has delivered the message. It is useful in multi-hops networks to know which node has delivered the message.
- **naS**
16-bit Source Network Address. It is filled by the receiver, and it specifies the address of the node which has delivered the message. It is useful in multi-hops networks to know which node has delivered the message.

- **macOL & macOH**

64-bit Origin MAC Address. It is filled by the receiver, and it specifies the address of the node which has sent originally the packet. It is useful in multi-hops networks to know which node has created and sent the message.

- **naO**

16-bit Origin Network Address. It is filled by the receiver, and it specifies the address of the node which has sent originally the packet. It is useful in multi-hops networks to know which node has created and sent the message.

- **niO**

Origin Node Identifier. It is filled by the receiver, and it specifies the node identifier of the node which has sent originally the packet. It is useful in multi-hops networks to know which node has created and sent the message.

- **RSSI**

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

In a fragmented packet, it contains the average of all received fragments RSSI.

- **address_typeS**

Address type used to send the packet by the transmitter. In ZigBee, it is always set to '1'.

- 0 : 16-bit transmission
- 1 : 64-bit transmission

- **typeSourceID**

Source ID type used to send the packet. It is filled by the transmitter and it specifies the ID at application level.

- 0 : 16-bit Network Address ID
- 1 : 64-bit MAC Address ID
- 2 : Node Identifier ID

- **numFragment**

Number of the fragment indicating the position in the fragmented packet. It is filled by the transmitter to indicate if the sent fragment is the first, second or last one.

- **endFragment**

Specifies if the sent fragment is the last fragment of a packet. It is filled by the transmitter.

- **time**

Specifies the time when the first fragment was received.

- **deliv_status**

Delivery status returned when transmitting a packet. It specifies the success or failure reason of the last packet sent:

- 0x00 : Success
- 0x02 : CCA failure
- 0x15 : Invalid Destination Endpoint
- 0x21 : Network ACK failure
- 0x22 : Not Joined to the Network
- 0x23 : Self-addressed
- 0x24 : Address not found
- 0x25 : Route not found

- **discov_status**
Discovery stats returned when transmitting a packet. It specifies the processes executed to find the destination node:
 - 0x00 : No Discovery Overhead
 - 0x01 : Address Discovery
 - 0x02 : Route Discovery
 - 0x03 : Address and Route Discovery
- **true_naD**
16-bit Network Address the packet was delivered to (if success). If not success, this address matches the 16-bit address given in the transmission packet.
- **retries**
The number of application transmission retries that took place.

5.2. Maximum payloads

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

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

Figure 10: Maximum Payloads Size

5.2.1. 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(); // Get Maximum Payload Bytes
}
```

Related Variables

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

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
0	-8dBm
1	-4dBm
2	-2dBm
3	0dBm
4	+2dBm

Figure 11: Power Output Levels

NOTE: XBee-PRO maximum value is +17dBm and XBee-PRO(International) is +10dBm

NOTE2: 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:

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

Graphics about transmission power are exposed next:

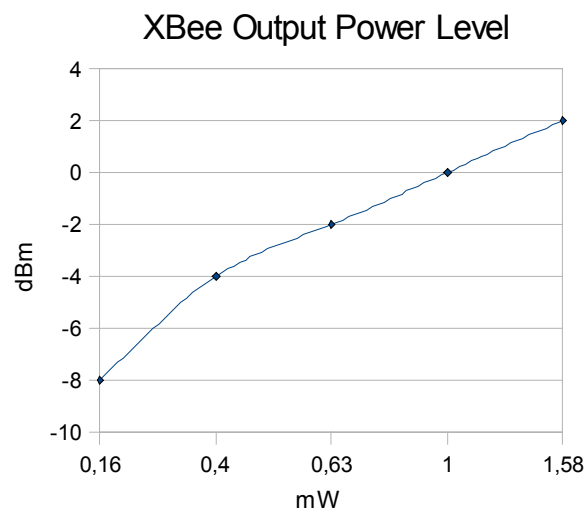


Figure 12: XBee Output Power Level

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

6.2. Power Mode

Enables or disables boost mode. This mode improves the receiver sensitivity in 1dB and increases the transmit power in 2dB.

Example of use

```
{
  xbeeZB.setPowerMode(0); // Set Power Mode to disable Boost Mode
  xbeeZB.getPowerMode(); // Get Power Mode
}
```

Related Variables

xbeeZB.powerMode → stores if boost mode is enabled or disabled

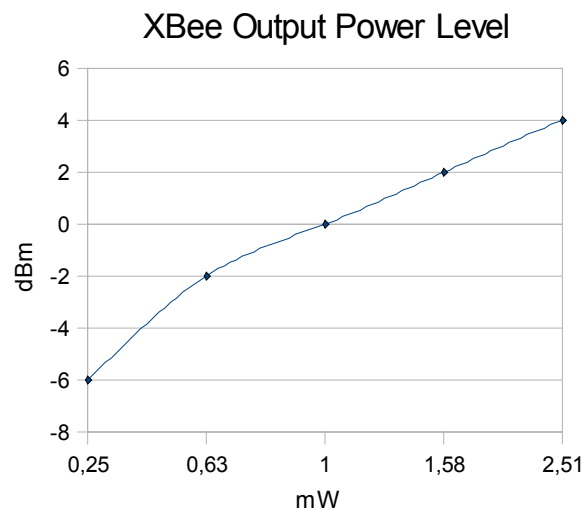


Figure 13: XBee Output Power Level (Boost Mode ON)

6.3. 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(); // Get the Receive Signal Strength Indicator
}
```

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.

7. Radio Channels

XBee ZigBee module works on 2.4GHz band, so there are 16 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.

7.1. 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); // Set all channels to scan
  xbeeZB.getScanningChannels(); // Get scanned channel list
}
```

Related Variables

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

7.2. 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

8. Connectivity

8.1. 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.

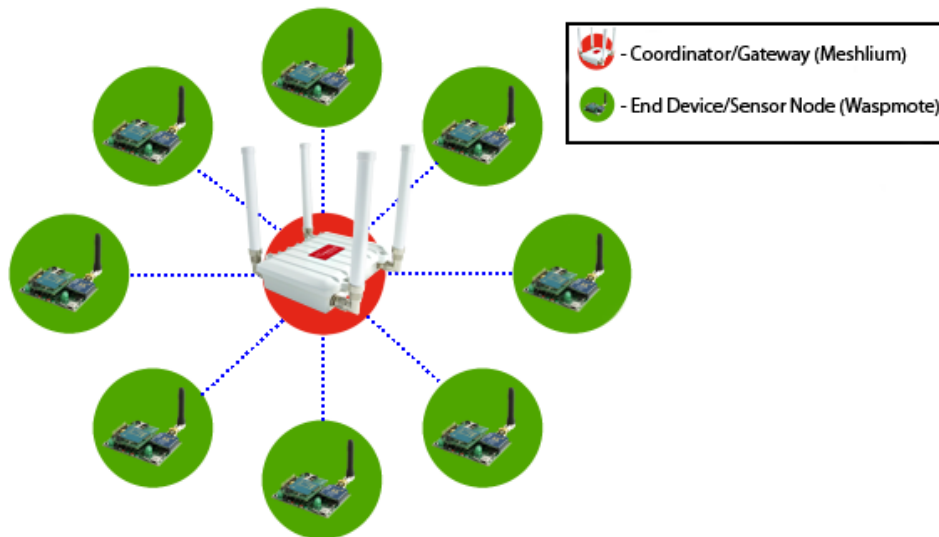


Figure 14: Star Topology

- **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.

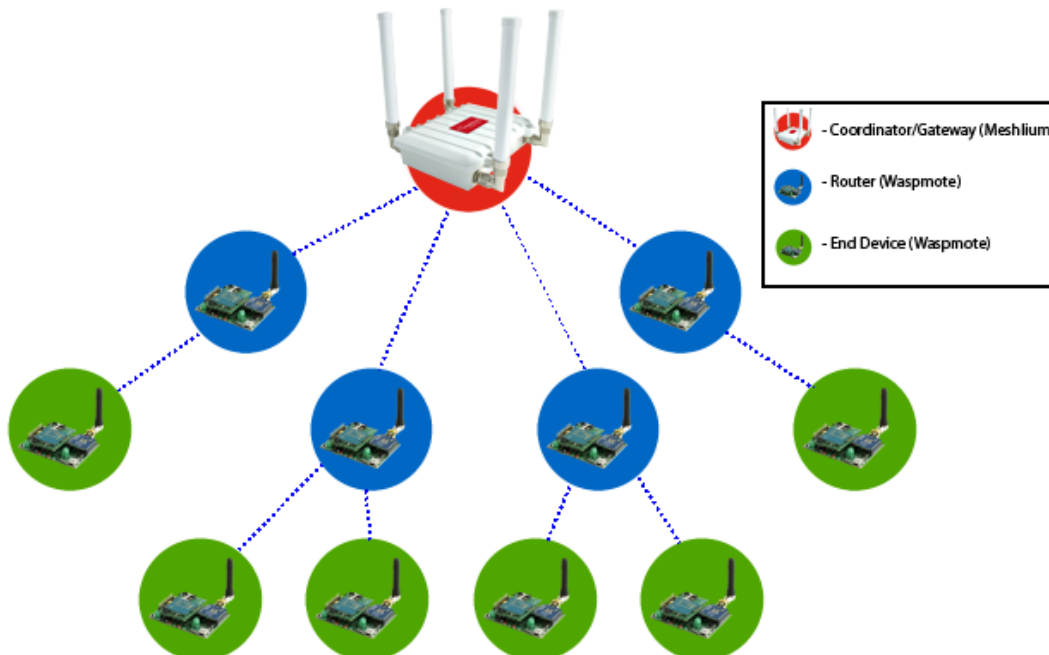


Figure 15: Tree Topology

ZigBee is aimed at working on star or tree. 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 (p2p) mesh topology, it is recommended to choose another protocol as DigiMesh.

8.2. ZigBee Architecture

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

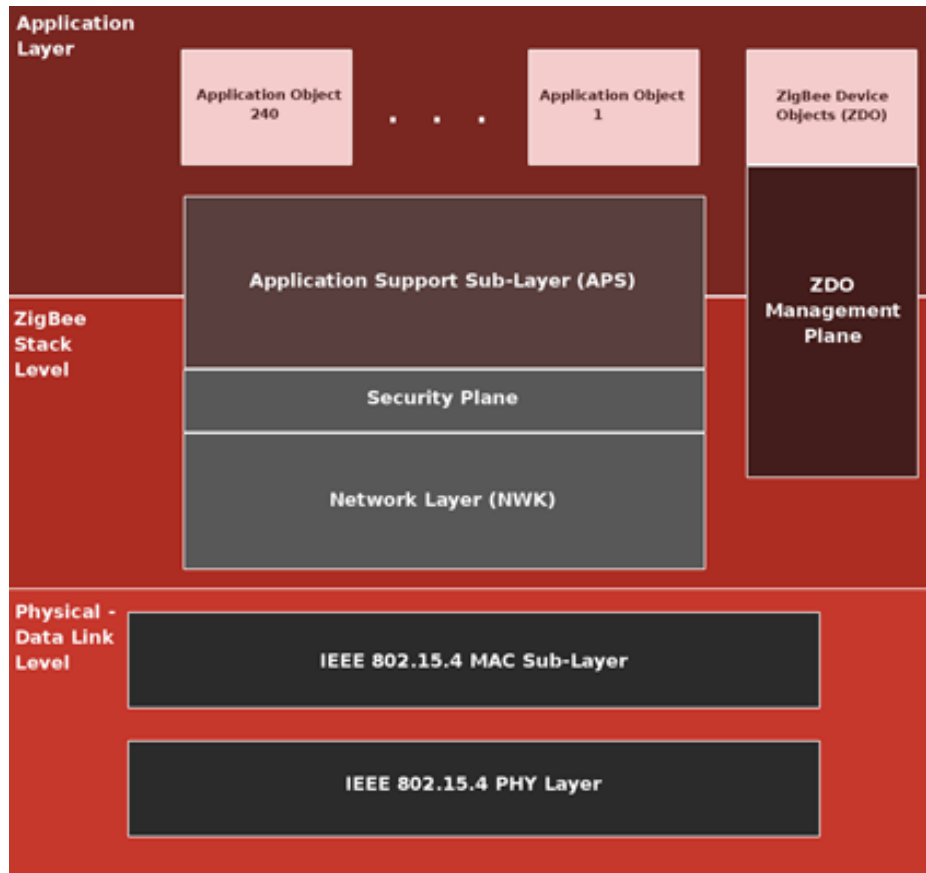


Figure 16: ZigBee Stack

8.2.1. 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.

8.2.2. 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.

8.2.3. 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.

8.2.4. 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()` (see chapter 2 for further information).

8.2.5. 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 `encryptionMode()`, `setLinkKey()`, `setNetworkKey()` or `setEncryptionOptions()` (see chapter 14 for further information).

8.2.6. 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).

8.2.7. Service Access Points

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

8.2.8. 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).

8.2.9. 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.

8.2.10. 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).

8.2.11. 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.

8.2.12. 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).

8.2.13. Waspote 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
}
```

8.3. 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.

8.3.1. 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.

8.3.2. 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.

8.4. Connections

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

8.4.1. 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 0x0000FFFF. 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 8.6 for further information.

8.4.2. 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.

8.4.3. 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.

8.4.4. 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

8.5. Discovering and Searching Nodes

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

8.5.1. 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:

```
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=Coord
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**
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 for application layer addressing.
- **MID**
Manufacturer ID. ID set by the manufacturer to identify the module.
- **RSSI**
Not returned in ZigBee.

To store the found nodes, an array called 'scannedBrothers' has been created. It is an array of structures 'Node'. A constant called 'MAX_BROTHERS' is defined to specify the maximum number of found brothers. A variable called 'totalScannedBrothers' is defined to indicate the number of brothers discovered. Using this variable as the index in the 'scannedBrothers' array allows to read the information discovered about each node.

Example of use:

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

Related variables

xbeeZB.totalScannedBrothers → stores the number of brothers discovered.

xbeeZB.scannedBrothers → 'Node' structure array that stores in each position the info related to each found node.
For example, xbeeZB.scannedBrothers[0].DT should store the Device Type of the first found node.

8.5.2. 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

```
{  
  packetXBee* paq_sent;  
  xbeeZB.nodeSearch("forrestNode-01",14,paq_sent); // '14' is the length of the string  
}
```

Related variables

paq_sent → Stores the 16-bit and 64-bit addresses of the searched. For example
paq_sent->MY[0] & paq_sent->MY[1] → stores the 16-bit NA of the searched node.

8.5.3. 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("forrestNode-01"); // Performs a ND to that specific node
}
```

Related variables

xbeeZB.totalScannedBrothers → stores the number of brothers discovered. In this case its value will be '1'

xbeeZB.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

8.6. 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 5) a packet usually needs to be fragmented. In order to manage this process, an application header has been created. This application header is sent inside RF Data, following the API Frame Structure.

8.6.1. API Frame Structure

The structure used to transmit packets via RF is specified by the used modules. The application header has been added to that defined structured as shown below:

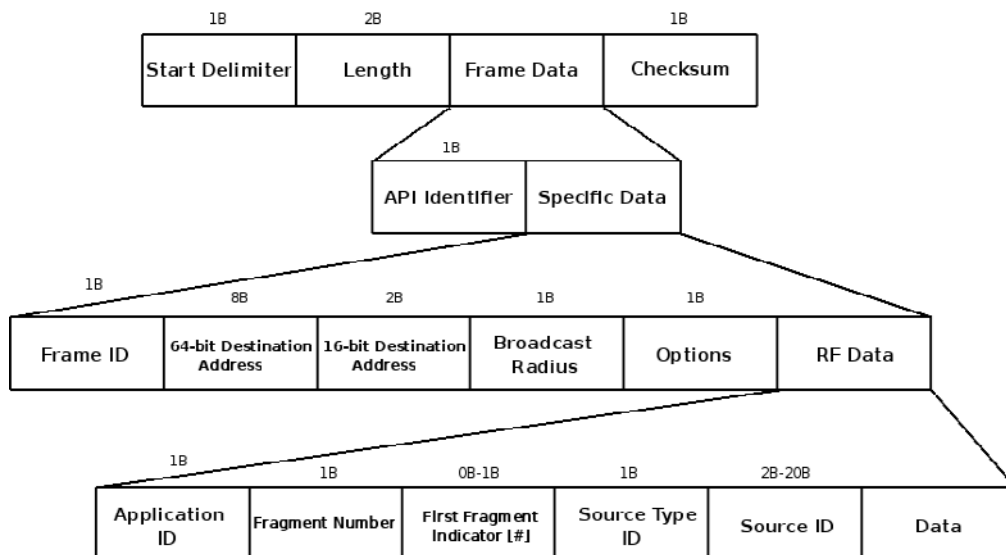


Figure 17: API Frame Structure

8.6.2. Application Header

This application header is filled by the transmitter (internal API operation) and it is used by the receiver to treat the packet or fragment. It is sent in the data field, so the maximum payload is reduced in a variable length (depending on the source ID type chosen).

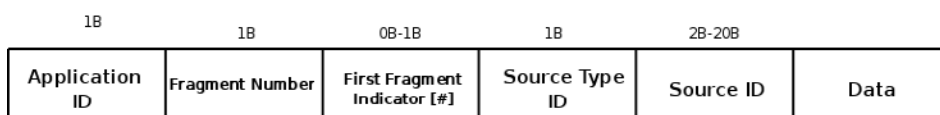


Figure 18: Application Header

To add some application level features to the receiver, some fields have been added like Application ID or Source ID. Source ID identifies the origin node which created the packet. It is useful in multi-hops networks, to identify the origin node to an answer. To identify this node, three Source IDs can be chosen:

- MAC Address: 64-bit MAC Source Address
- Network Address: 16-bit Network Address
- Node Identifier: 20-byte max string NI

The different fields of this header and its structure inside the API Frame Structure are:

- Application ID: it specifies the ID at application level. It allows to identify a single packet among several packets in the receiver.
- Fragment Number: it specifies the position of the fragment in its packet. The first fragment shows the total number of fragments of the packet.
- First Fragment Indicator [#]: optional field that is included in the first fragment to indicate that it is the first fragment.
- Source Type ID: it specifies the source ID type chosen. The possibilities are MAC, 16-bit or NI.
- Source ID: it specifies the source ID of the origin node. It is related to the previous field.
- Data: it stores the data sent in each fragment or packet.

8.6.3. Fragmentation

When the data length of a packet exceeds the maximum payload, the packet has to be fragmented to reach the destination. If the packet is not fragmented, the module will discard it before transmission.

NOTE: Due to restricted memory in Wasp mote, the maximum recommended data in a packet is **1500 Bytes (to be fragmented)**.

8.6.4. Structure packetXBee

It is the structure used for the packet to send. It was explained in chapter 5.

The process to send a packet is explained next:

1. A structure 'packetXBee' is created to contain the packet to send.
2. That structure is filled with the correct data into the corresponding fields.
3. The API function responsible of sending data is called, using the previously created structure as the input.

The API function takes care of the rest of the process, returning a success or failure to the process.

8.6.5. Sending without API header

There is a function to send raw data without the API header. This mode has been designed to send simple data to a gateway when the API features are not necessary.

Broadcast and Unicast transmissions are supported, as well as encryption.

8.6.6. Examples

To simplify the examples and make them easy to read, there are some lines that don't work exactly in C/C++.

- **Sending a packet of 50 bytes without API header**

```
{
    char* data;
    for(int c=0;c<50;c++) // Set the data
    {
        data[c]='A';
    }
    xbeeZB.send("0013A2004030F66A",data);
}
```

- Sending a 50-bytes data packet. Unicast. 16-bit Network Address Unknown. 16-bit NA Source ID Type.**

```
{
    packetXBee* paq_sent; // create packet to send
    char* data;
    paq_sent=(packetXBee*) calloc(1,sizeof(packetXBee));
    paq_sent->mode=UNICAST; // set Unicast mode
    paq_sent->MY_known=0; // set 16-bit NA unknown
    paq_sent->packetID=0x52; // set ID application level
    paq_sent->opt=0x00; // set options. No option selected.
    for(int c=0;c<50;c++) // Set the data
    {
        data[c]='A';
    }
    xbeeZB.setOriginParams(paq_sent, "1221", MY_TYPE);
    xbeeZB.setDestinationParams(paq_sent, "0013A2004030F686", data, MAC_TYPE, DATA_ABSOLUTE);
    state=xbeeZB.sendXBee(paq_sent); // Call function responsible to send data
    if(!state)
    {
        delivery_status=paq_sent->deliv_status;
        discovery_status=paq_sent->discov_status;
        delivered_address=paq_sent->true_naD;
        retries=paq_sent->retries;
    }
    free(paq_sent);
    paq_sent=NULL;
}
```

The data field in the API frame generated by the function 'sendXBee' is:

Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
0x52	1	#	0	0x12 0x21	A A A ... A A (50 Bytes)

Figure 19: API Application Header

- Sending a 50-bytes data packet. Unicast. 16-bit Network Address Known. 64-bit MAC Source ID Type.**

```
{
    packetXBee* paq_sent; // create packet to send
    char* data;
    paq_sent=(packetXBee*) calloc(1,sizeof(packetXBee));
    paq_sent->mode=UNICAST; // set Unicast mode
    paq_sent->MY_known=1; // set 16-bit NA known
    paq_sent->packetID=0x52; // set ID application level
    paq_sent->opt=0x00; // set options. No option selected.
    for(int c=0;c<50;c++) // Set the data
    {
        data[c]='A';
    }
    xbeeZB.setOriginParams(paq_sent, "0013A2004030F66A", MAC_TYPE);
    xbeeZB.setDestinationParams(paq_sent, "1221", data, MY_TYPE);
    xbeeZB.setDestinationParams(paq_sent, "0013A2004030F686", data, MAC_TYPE, DATA_ABSOLUTE);
    state=xbeeZB.sendXBee(paq_sent); // Call function responsible to send data
    if(!state)
    {
        delivery_status=paq_sent->deliv_status;
        discovery_status=paq_sent->discov_status;
        delivered_address=paq_sent->true_naD;
        retries=paq_sent->retries;
    }
    free(paq_sent);
    paq_sent=NULL;
}
```

The data field in the API frame generated by the function 'sendXBee' will be:

Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
0x52	1	#	1	0x00 0x13 0xA2 0x00 0x40 0x30 0xF6 0x86	A A A ... A A (50 Bytes)

Figure 20: API Application Header

- Sending a 50-bytes data packet. Broadcast. Node Identifier Source ID Type.**

```
{
    packetXBee* paq_sent; // create packet to send
    char* data;
    paq_sent=(packetXBee*) calloc(1,sizeof(packetXBee));
    paq_sent->mode=BROADCAST; // set Broadcast mode
    paq_sent->MY_known=0; // set 16-bit NA unknown
    paq_sent->packetID=0x52; // set ID application level
    paq_sent->opt=0x00; // set options. No option selected.
    for(int c=0;c<50;c++) // Set the data
    {
        data[c]='A';
    }
    xbeeZB.setOriginParams(paq_sent, "forrestNode-01", NI_TYPE);
    xbeeZB.setDestinationParams(paq_sent, "000000000000FFFF", data, MAC_TYPE, DATA_ABSOLUTE);
    state=xbeeZB.sendXBee(paq_sent); // Call function responsible to send data
    if(!state)
    {
        delivery_status=paq_sent->deliv_status;
        discovery_status=paq_sent->discov_status;
        delivered_address=paq_sent->true_naD;
        retries=paq_sent->retries;
    }
    free(paq_sent);
    paq_sent=NULL;
}
```

The data field in the API frame generated by the function 'sendXBee' will be:

Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
0x52	1	#	2	forrestNode-01	A A A ... A A (50 Bytes)

Figure 21: API Application Header

- Sending a 200-bytes data packet. Unicast 16-bit Network Address Unknown. 16-bit Source ID Type.**

```
{
    packetXBee* paq_sent; // create packet to send
    char* data;
    paq_sent=(packetXBee*) calloc(1,sizeof(packetXBee));
    paq_sent->mode=UNICAST; // set Unicast mode
    paq_sent->MY_known=0; // set 16-bit NA unknown
    paq_sent->packetID=0x52; // set ID application level
    paq_sent->opt=0x00; // set options. No option selected.
    for(int c=0;c<200;c++) // Set the data
    {
        data[c]='A';
    }
    xbeeZB.setOriginParams(paq_sent, "1221", MY_TYPE);
    xbeeZB.setDestinationParams(paq_sent, "0013A2004030F686", data, MAC_TYPE, DATA_ABSOLUTE);
    state=xbeeZB.sendXBee(paq_sent); // Call function responsible to send data
    if(!state)
    {
```

```

delivery_status=paq_sent->deliv_status;
discovery_status=paq_sent->discov_status;
delivered_address=paq_sent->true_nAd;
retries=paq_sent->retries;
}
free(paq_sent);
paq_sent=NULL;
}

```

Due to the large amount of data, the packet needs to be fragmented before being sent. The maximum data payload is 74Bytes (see chapter 5) and the application header will occupy a maximum of 6 Bytes (in the first fragment), so there will be three fragments, sending 68Bytes(data) in first fragment, 69Bytes(data) in second fragment and 63Bytes(data) in third fragment.

	Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
First Fragment	0x52	3	#	0	0x12 0x21	A A A ... A A (68 Bytes)
Second Fragment	0x52	2	0	0x12 0x21		A A A ... A A (69 Bytes)
Third Fragment	0x52	1	0	0x12 0x21		A A A ... A A (63 Bytes)

Figure 22: API Application Header

- Sending a 50-bytes data packet. Application Layer Addressing. 64-bit Source ID Type**

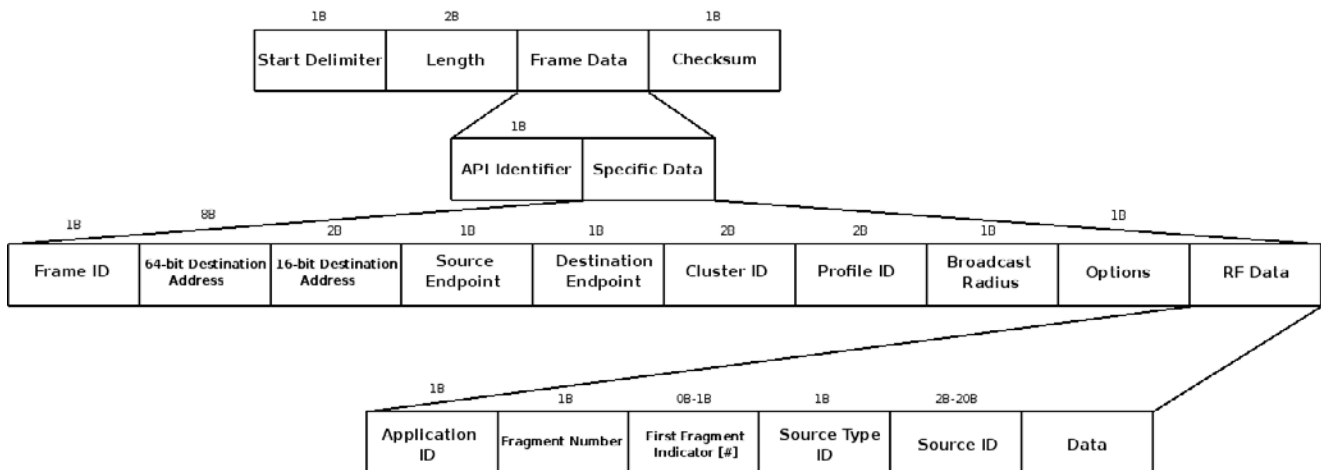


Figure 23: API Frame Extended Structure

When using application layer addressing, the modules use the structure above to send the packets. Application header is inserted in RF Data field as in device addressing examples.

```

{
packetXBee* paq_sent; // create packet to send
char* data;
paq_sent=(packetXBee*) calloc(1,sizeof(packetXBee));
paq_sent->mode=CLUSTER; // set Application Layer Addressing
paq_sent->SD=0x12; // set source endpoint
paq_sent->DE=0x13; // set destination endpoint
    paq_sent->CID={0x14,0x15}; // set Cluster Identifier **
    paq_sent->PID={0x16,0x17}; // set Profile ID **
paq_sent->MY_known=1; // set 16-bit NA known
paq_sent->packetID=0x52; // set ID application level

```

```

paq_sent->opt=0x00; // set options. No option selected.
for(int c=0;c<50;c++) // Set the data
{
    data[c]='A';
}
xbeeZB.setOriginParams(paq_sent, "0013A2004030F66A", MAC_TYPE);
xbeeZB.setDestinationParams(paq_sent, "1221", data, MY_TYPE, DATA_ABSOLUTE);
xbeeZB.setDestinationParams(paq_sent, "0013A2004030F686", data, MAC_TYPE, DATA_ABSOLUTE);
state=xbeeZB.sendXBee(paq_sent); // Call function responsible to send data
if(!state)
{
    delivery_status=paq_sent->deliv_status;
    discovery_status=paq_sent->discov_status;
    delivered_address=paq_sent->true_nad;
    retries=paq_sent->retries;
}
free(paq_sent);
paq_sent=NULL;
}

** Code Simplification. Arrays must be filled one by one.

```

When using application layer addressing, both destination address(MAC & NA) and application address(SD, DE, CID, PID) should be set in the packet.

A explicit addressing frame is sent when using application layer addressing which is a bit different from normal transmit frame and includes all the fields above.

The data field in the API frame generated by the function 'sendXBee' will be:

Application ID	Fragment Number	First Fragment Indicator [#]	Source Type ID	Source ID	Data
0x52	1	#	1	0x00 0x13 0xA2 0x00 0x40 0x30 0xF6 0x86	A A A ... A A (50 Bytes)

Figure 24: API Application Header

8.7. 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, a multidimensional array of 'matrix' structures pointers, an array of 'index' structures pointers and an array of 'packetXBee' structures pointers are created. The size of these arrays is defined by some constants in compilation time, so it is necessary to adequate its value to each scenario before compile the code:

- **MAX_FINISH_PACKETS:** max number of finished packets pending of treatment. It specifies the size of the array of finished packets.
- **MAX_FRAG_PACKETS:** max number of fragments per packet. It specifies the number of columns in the multidimensional array of pending fragments. The number of rows is specified by **MAX_FINISH_PACKETS**.

When a packet or fragment is received, the algorithm used in reception API function is:

1. Check if the fragment is a new packet or belongs to an existing packet. If it is a new packet, an structure 'index' is created and linked to the array index previously created. The application level information is stored in that structure. If it belongs to an existing packet, it doesn't store any information.
2. Create an structure 'matrix' and link it to the corresponding position in the multidimensional matrix array. The position is calculated upon the information stored in the 'index' array.
3. Store the data in the corresponding position of the matrix.

4. Check if the packet is complete. If the packet is complete, a 'packetXBee' structure is created and linked to the corresponding position in the array of finished packets. The different fragments are ordered and copied to this structure, as the application level information. The 'index' and row of 'matrix' are released, to free memory. If the packet is not complete, goes to next step.
5. Exit the function.

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 of reading packets can read more than one fragment, but the XBee module may overflow its buffer, so it is recommended to read packets one by one.

When a packet is completed, it will be stored in a finished packets array called 'packet_finished'. This array should be used by the application to read the received packet. A variable called 'pos' is used to know if there are pending received packets : if 'pos'=0, there is no packet available ; if 'pos'>0, there will be as many pending packets as its value (pos=3, 3 pending packets).

8.7.1. Examples

- **Received 50-bytes data packet. Unicast. 16-bit NA Source ID Type**

```
{
  state=xbeeZB.treatData(); // Read a packet when XBee has noticed it to us
  if( xbeeZB.pos>0 )
  {
    /* Available info: (there are more available information) */
    network_address[0]=packet_finished[xbeeZB.pos-1]->naS[0];
    network_address[1]=packet_finished[xbeeZB.pos-1]->naS[1];
  }
}
```

Available Information

packet_finished[xbeeZB.pos-1]->naS : stores the 16-bit network address of the sender.
 packet_finished[xbeeZB.pos-1]->macSH : stores the 32 upper bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->macSL : stores the 32 lower bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->naO : stores the 16-bit network address of the origin sender.
 packet_finished[xbeeZB.pos-1]->mode : stores the transmission mode. Unicast in this case.
 packet_finished[xbeeZB.pos-1]->data : stores the data received. Max size 'MAX_DATA'.
 packet_finished[xbeeZB.pos-1]->packetID : stores the packet ID of the received message. 0x52 in this case.
 packet_finished[xbeeZB.pos-1]->typeSourceID : stores the source type ID used in the message.

After receiving the packet, due to there is only one fragment, it will be completed, so variable 'pos' will be greater than zero. This variable is used to index the array where the received packet is stored. The information that has been stored is indicated in the example.

- **Received 50-bytes data packet. Unicast. 64-bit MAC Address Source ID Type**

```
{
  state=xbeeZB.treatData(); // Read a packet when XBee has noticed it to us
  if( xbeeZB.pos>0 )
  {
    /* Available info: (there are more available information) */
    source_address[0]=packet_finished[xbeeZB.pos-1]->macSH[0];
    ...
    source_address[3]=packet_finished[xbeeZB.pos-1]->macSH[3];
  }
}
```

Available Information

packet_finished[xbeeZB.pos-1]->naS : stores the 16-bit network address of the sender.
 packet_finished[xbeeZB.pos-1]->macSH : stores the 32 upper bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->macSL : stores the 32 lower bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->macOH : stores the 32 upper bits of MAC origin sender address.

packet_finished[xbeeZB.pos-1]->macOL : stores the 32 lower bits of MAC origin sender address.
 packet_finished[xbeeZB.pos-1]->mode : stores the transmission mode. Unicast in this case.
 packet_finished[xbeeZB.pos-1]->data : stores the data received. Max size 'MAX_DATA'.
 packet_finished[xbeeZB.pos-1]->packetID : stores the packet ID of the received message. 0x52 in this case.
 packet_finished[xbeeZB.pos-1]->typeSourceID : stores the source type ID used in the message.

After receiving the packet, due to there is only one fragment, it will be completed, so variable 'pos' will be greater than zero. This variable is used to index the array where the received packet is stored. The information that has been stored is indicated in the example.

- **Received 50-bytes data packet. Broadcast. Node Identifier Source ID Type**

```

{
  state=xbeeZB.treatData(); // Read a packet when XBee has noticed it to us
  if( xbeeZB.pos>0 )
  {
    /* Available info: (there are more available information) */
    source_address[0]=packet_finished[xbeeZB.pos-1]->macSH[0];
    ...
    source_address[3]=packet_finished[xbeeZB.pos-1]->macSH[3];
  }
}

```

Available Information

packet_finished[xbeeZB.pos-1]->naS : stores the 16-bit network address of the sender.
 packet_finished[xbeeZB.pos-1]->macSH : stores the 32 upper bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->macSL : stores the 32 lower bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->niO : stores 20-byte max string used to Node Identifier of the origin sender
 packet_finished[xbeeZB.pos-1]->mode : stores the transmission mode. Unicast in this case.
 packet_finished[xbeeZB.pos-1]->data : stores the data received. Max size 'MAX_DATA'.
 packet_finished[xbeeZB.pos-1]->packetID : stores the packet ID of the received message. 0x52 in this case.
 packet_finished[xbeeZB.pos-1]->typeSourceID : stores the source type ID used in the message.

After receiving the packet, due to there is only one fragment, it will be completed, so variable 'pos' will be greater than zero. This variable is used to index the array where the received packet is stored. The information that has been stored is indicated in the example.

- **Received 200-bytes data packet. Unicast. 16-bit NA Source ID Type**

```

{
  state=xbeeZB.treatData(); // Read a packet when XBee has noticed it to us
  if( xbeeZB.pos>0 )
  {
    /* Available info: (there are more available information) */
    network_address[0]=packet_finished[xbeeZB.pos-1]->naS[0];
    network_address[1]=packet_finished[xbeeZB.pos-1]->naS[1];
  }
}

```

Available Information

packet_finished[xbeeZB.pos-1]->naS : stores the 16-bit network address of the sender.
 packet_finished[xbeeZB.pos-1]->macSH : stores the 32 upper bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->macSL : stores the 32 lower bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->naO : stores the 16-bit network address of the origin sender.
 packet_finished[xbeeZB.pos-1]->mode : stores the transmission mode. Unicast in this case.
 packet_finished[xbeeZB.pos-1]->data : stores the data received. Max size 'MAX_DATA'.
 packet_finished[xbeeZB.pos-1]->packetID : stores the packet ID of the received message. 0x52 in this case.
 packet_finished[xbeeZB.pos-1]->typeSourceID : stores the source type ID used in the message.

After receiving the first packet, due to there are three fragments, it will not be completed, so variable 'pos' will be zero. It is recommended to set a delay in the transmitter so as to the receiver doesn't receive more than one packet at once. When the three fragments have been received, variable 'pos' will change its value and the information in the 'packet_finished' array will be available.

NOTE: Due to memory restrictions, it is recommended when a pending packet is treated and it is not necessary to store it any more, to release this pointer. If don't, memory may overflow and crash the application. It is not possible to have more than 2700Bytes in a receiver node. It is referred to a single big packet or many smaller packets in parallel.

- **Received 50-bytes data packet. Application Layer Addressing. 64-bit Source ID Type**

```
{
  state=xbeeZB.treatData(); // Read a packet when XBee has noticed it to us
  if( xbeeZB.pos>0 )
  {
    /* Available info: (there are more available information) */
    source_address[0]=packet_finished[xbeeZB.pos-1]->macSH[0];
    ...
    source_address[3]=packet_finished[xbeeZB.pos-1]->macSH[3];
  }
}
```

Available Information

packet_finished[xbeeZB.pos-1]->naS : stores the 16-bit network address of the sender.
 packet_finished[xbeeZB.pos-1]->macSH : stores the 32 upper bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->macSL : stores the 32 lower bits of MAC sender address.
 packet_finished[xbeeZB.pos-1]->macOH : stores the 32 upper bits of MAC origin sender address.
 packet_finished[xbeeZB.pos-1]->macOL : stores the 32 lower bits of MAC origin sender address.
 packet_finished[xbeeZB.pos-1]->SD : stores the Source Endpoint of the origin sender
 packet_finished[xbeeZB.pos-1]->DE : stores the Destination Endpoint
 packet_finished[xbeeZB.pos-1]->CID : stores the Cluster Identifier
 packet_finished[xbeeZB.pos-1]->PID : stores the Profile Identifier
 packet_finished[xbeeZB.pos-1]->mode : stores the transmission mode. Unicast in this case.
 packet_finished[xbeeZB.pos-1]->data : stores the data received. Max size 'MAX_DATA'.
 packet_finished[xbeeZB.pos-1]->packetID : stores the packet ID of the received message. 0x52 in this case.
 packet_finished[xbeeZB.pos-1]->typeSourceID : stores the source type ID used in the message.

After receiving the packet, due to there is only one fragment, it will be completed, so variable 'pos' will be greater than zero. This variable is used to index the array where the received packet is stored. The information that has been stored is indicated in the example.

When using application layer addressing, a special frame is returned by the module called explicit addressing frame. To allow the module to receive application layer addressing packets, API options should be changed. This explicit addressing frame returns the information related to endpoint, cluster and profile used in the application layer communication.

9. Starting a Network

9.1. 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 13.

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.

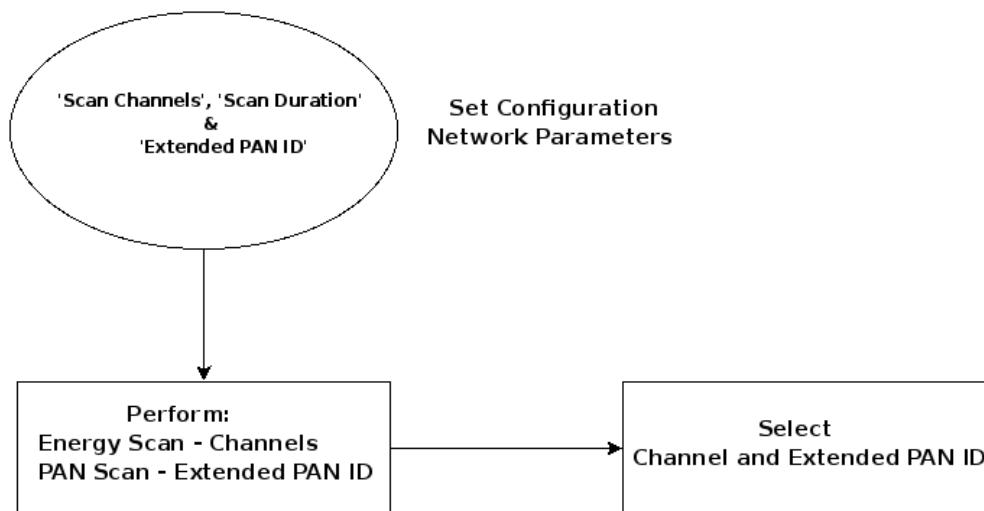


Figure 25: Coordinator Startup

9.2. Coordinator wake up process from Reset

When a coordinator wakes up from a reset or power cycle, it 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 does not match the saved channel, PAN ID or security, it will leave the PAN and will start in other PAN matching the configuration parameters.

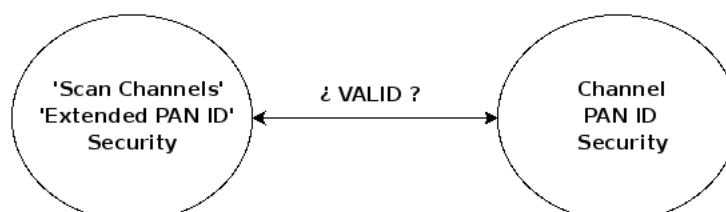


Figure 26: Coordinator Wake Up Process

9.3. Related Parameters

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

9.3.1. 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

9.3.2. 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

9.4. Examples

9.4.1. Coordinator Startup Process. Random Extended PAN ID. All channels to scan

```
{
  panid={0,0,0,0,0,0,0,0};
  xbeeZB.setPAN(panid); // Set PANID to '0'. Coordinator will select a random PAN ID
  xbeeZB.setScanningChannels(0xFF,0xFF); // Set List of Channels to scan.
  xbeeZB.setDurationEnergyChannels(3); // Set exponent of scan channel duration
  xbeeZB.getAssociationIndication(); // Check if creating process success
  if(!xbeeZB.associationIndication)
  {
    /* Association Successful
  }
}
```

Once these configuration parameters have been set, the Coordinator will scan to select the channel and PAN ID. When the Coordinator finishes the process, 'associationIndication' will change to '0x00'.

Available Information

xbeeZB.associationIndication → stores the status of the last creating a network process
xbeeZB.extendedPAN → stores the 64-bit Extended Operating PAN ID
xbeeZB.operatingPAN → stores the 16-bit Operating PAN ID
xbeeZB.channel → stores the channel selected

9.4.2. Coordinator Startup Process. Selected Extended PAN ID. Only some channels to scan

```
{
  panid={0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
  xbeeZB.setPAN(panid); // Set PANID to the selected. Coordinator will select this PAN ID
  xbeeZB.scanChannels(0x00,0xFF); // Set List of Channels to scan.
  xbeeZB.setDurationEnergyChannels(3); // Set exponent of scan channel duration
  xbeeZB.getAssociationIndication(); // Check if creating process success
  if(!xbeeZB.associationIndication)
  {
    /* Association Successful
  }
}
```

Once these configuration parameters have been set, the Coordinator will scan to select the channel and PAN ID. When the Coordinator finishes the process, 'associationIndication' will change to '0x00'.

Available Information

xbeeZB.associationIndication → stores the status of the last creating a network process
xbeeZB.extendedPAN → stores the 64-bit Extended Operating PAN ID
xbeeZB.operatingPAN → stores the 16-bit Operating PAN ID
xbeeZB.channel → stores the channel selected

10. 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.

10.1. 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 'Extended PAN ID' of the device is valid depending on the router 'Extended PAN ID'. If router 'Extended 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 policy 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.

10.1.1. Example

```
{
  panid={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
  xbeeZB.setPAN(panid); // Set PANID to '0'. Router will join any PAN ID
  xbeeZB.setScanningChannels(0xFF,0xFF); // Set List of Channels to scan. All channels scanned
  xbeeZB.setDurationEnergyChannels(3); // Set exponent of scan channel duration
  xbeeZB.getAssociationIndication(); // Check if creating process success
  if(!xbeeZB.associationIndication)
  {
    /* Association Successful
  }
}
```

Once these configuration parameters have been set, Router will start scans to find a valid PAN.

Available Information

xbeeZB.associationIndication → stores the status of the last creating a network process
xbeeZB.extendedPAN → stores the 64-bit Extended Operating PAN ID
xbeeZB.operatingPAN → stores the 16-bit Operating PAN ID
xbeeZB.channel → stores the channel selected

10.2. 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 'Extended PAN ID' of the device is valid depending on the router 'Extended PAN ID'. If router 'Extended 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.

10.2.1. Example

```
{
  panid={0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
  xbeeZB.setPAN(panid); // Set PANID. End Device will only join a PAN with that PAN ID.
  xbeeZB.scanChannels(0x00,0xFF); // Set List of Channels to scan.
  xbeeZB.setDurationEnergyChannels(3); // Set exponent of scan channel duration
  xbeeZB.getAssociationIndication(); // Check if creating process success
  if(!xbeeZB.associationIndication)
  {
    /* Association Successful
  }
}
```

Once these configuration parameters have been set, End Device will start scans to find a valid PAN.

Available Information

xbeeZB.associationIndication → stores the status of the last creating a network process
 xbeeZB.extendedPAN → stores the 64-bit Extended Operating PAN ID
 xbeeZB.operatingPAN → stores the 16-bit Operating PAN ID
 xbeeZB.channel → stores the channel selected

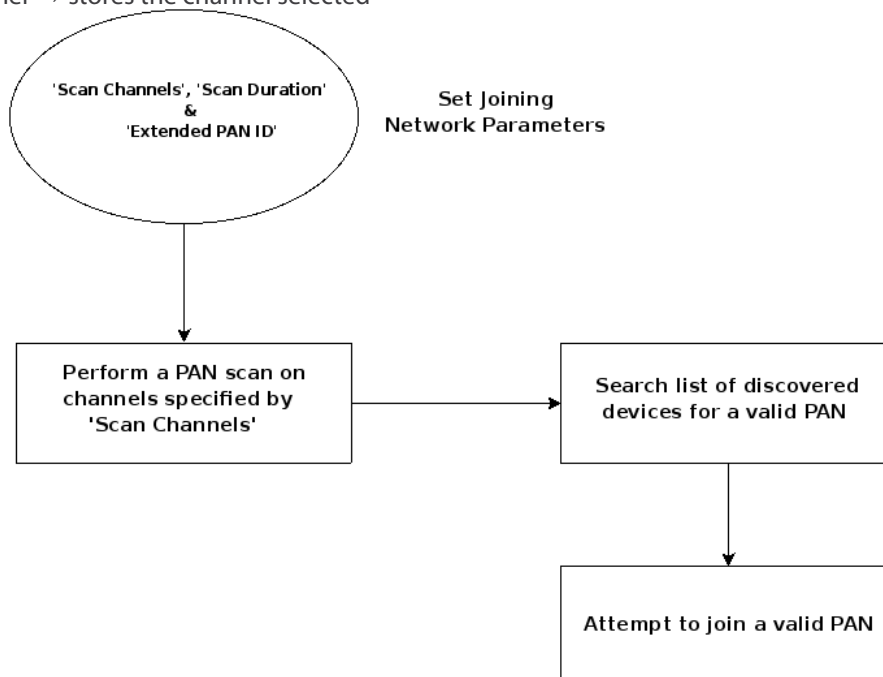


Figure 27: Router / End Device Joining Process

10.3. 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.

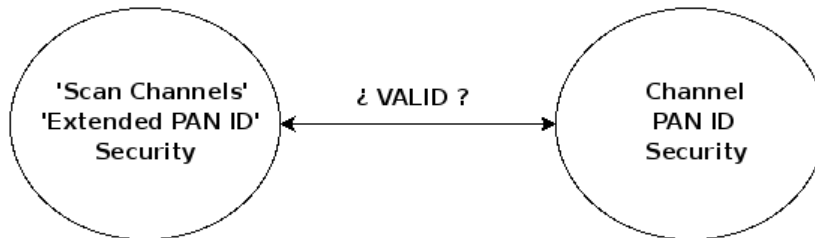


Figure 28: Router / End Device Coming up from Reset

10.4. 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.

10.4.1. 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.

10.4.2. 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.

10.5. Related Parameters

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

10.5.1. 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(); // Get Association Indication Status
}
```

Related Variables

xbeeZB.associationIndication → stores the Association Indication Status

10.5.2. 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 Routes. 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(); // Get the number of remaining children available
}
```

Related Variables

xbeeZB.remainingChildren → stores the number of remaining children available

10.5.3. Channel Verification

Specifies if Channel Verification is enabled or disabled. If Channel Verification is enabled, routers and end devices will behave as explained previously. If it is disabled, routers will continue working even if a coordinator is not detected.

Example of use:

```
{
  xbeeZB.setChannelVerification(1); // Enable Channel Verification
  xbeeZB.getChannelVerification(); // Get if Channel Verification is enabled or disabled
}
```

Related Variables

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

10.5.4. 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); // Enable Join Notification
  xbeeZB.getJoinNotification(); // Get if Join Notification is enabled or disabled
}
```


Related Variables

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

10.6. Node Discovery

Once these parameters have been set, the node is a part of the network, so the first task is to discover the other nodes in the network.

NOTE: Node Discovery process can not be executed in End Devices. No error will occur but no response will be received.

Some parameters are involved in a node discovery process.

10.6.1. Node Discovery

Performs a node discovery returning the found brothers in the network stored in an array of structures that contain information about the nodes.

Example of use:

```
{
  xbeeZB.scanNetwork(); // Discovery nodes
  if(xbeeZB.totalScannedBrothers>0)
  {
    /* Available info (There are more available information) */
    network_address[0]=scannedBrothers[totalScannedBrothers-1].MY[0];
    network_address[1]=scannedBrothers[totalScannedBrothers-1].MY[1];
  }
}
```

Available Information

scannedBrothers[totalScannedBrothers-1].MY → stores 16-bit NA of each module
scannedBrothers[totalScannedBrothers-1].SH → stores the 32 upper bits of MAC address
scannedBrothers[totalScannedBrothers-1].SL → stores the 32 lower bits of MAC address
scannedBrothers[totalScannedBrothers-1].NI → stores the Node Identifier
scannedBrothers[totalScannedBrothers-1].PMY → stores the 16-bit NA of its parent
scannedBrothers[totalScannedBrothers-1].DT → stores the device type(C, R or E)
scannedBrothers[totalScannedBrothers-1].ST → stores the status. Reserved by module.
scannedBrothers[totalScannedBrothers-1].MID → stores the 16-bit manufacturer ID
scannedBrothers[totalScannedBrothers-1].PID → stores the 16-bit Profile ID

10.6.2. Node Discovery Time

Is the amount of time a node will wait for responses from other nodes when performing a ND.

Example of use:

```
{
  uint8_t time[2]={0x19,0x00}; // 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 → stores the time a node will wait for responses.

10.6.3. 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(); // 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.

11. Rebooting a Network

11.1. 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
}
```

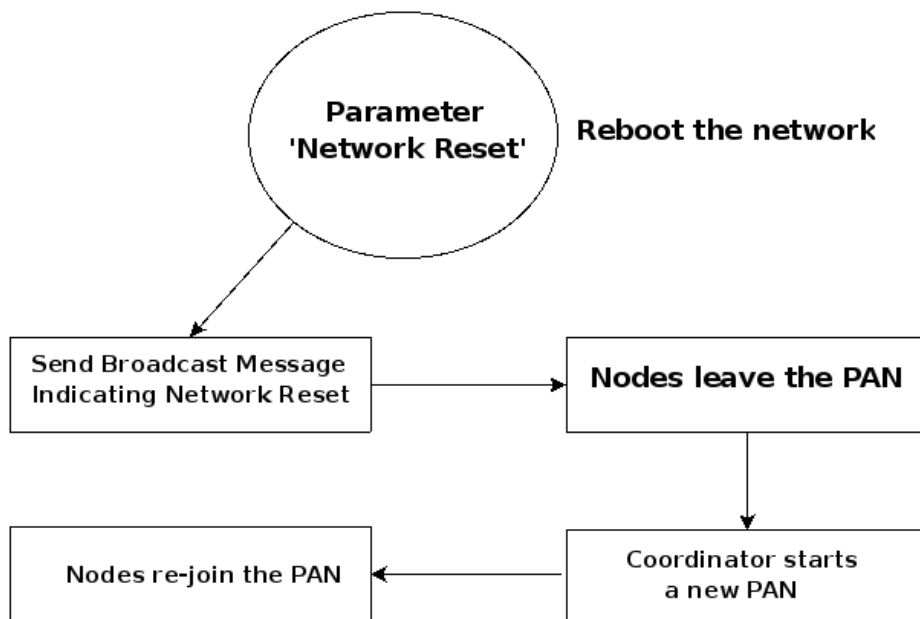


Figure 29: Network Reset Process

12. Sleep Options

12.1. Sleep Modes

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

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

12.1.1. Pin Sleep

This mode is used to wake and sleep the module using a hardware pin. If the module has joined a network, it will sleep when pin was asserted, after ending any transmission or reception in process. If 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.

12.1.2. 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.

12.2. Sleep Parameters

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

12.2.1. 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); // Set Sleep Mode to Pin Sleep
  xbeeZB.getSleepMode(); // Get the Sleep Mode used
}
```

Related Variables

xbeeZB.sleepMode → stores the sleep mode in a module

12.2.2. 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

12.2.3. 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); // Set time the module remains awake
}
```

Related Variables

xbeeZB.awakeTime → stores the time the module remains awake

12.2.4. 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); // Set number of sleep periods
}
```

Related Variables

xbeeZB.periodSleep → stores the number of sleep periods

12.2.5. 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); // Set always awake during 'Time Before Sleep' time.
}
```

Related Variables

xbeeZB.sleepOptions → stores the sleep option chosen

12.3. ON/OFF Mode

In addition to the XBee sleep modes, Wasmote provides the feature of controlling the power state with a digital switch. This means that using one function included in Wasmote API, any XBee module can be powered up or down (0uA).

Example of use:

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

13. Synchronizing the Network

13.1. 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.

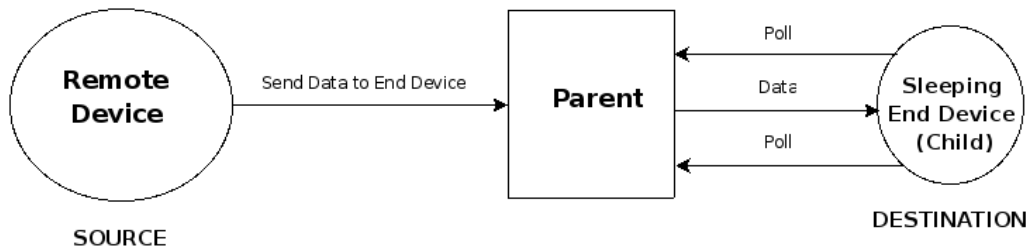


Figure 30: RF Data Sending to Sleeping End Device

13.2. 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.

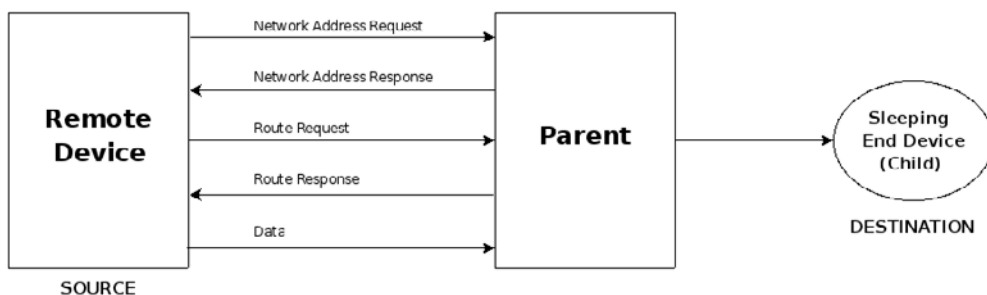


Figure 31: Determining Sleeping End Device Address and Route

13.3. 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 for 30 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.

14. Security and Data Encryption

14.1. 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.

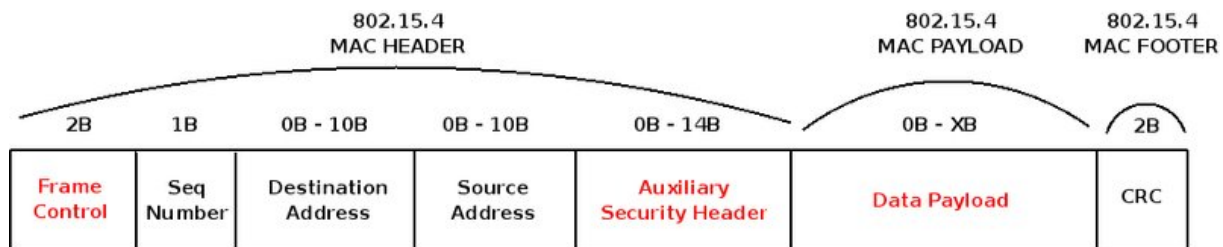


Figure 32: IEEE 802.15.4 Frame

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.

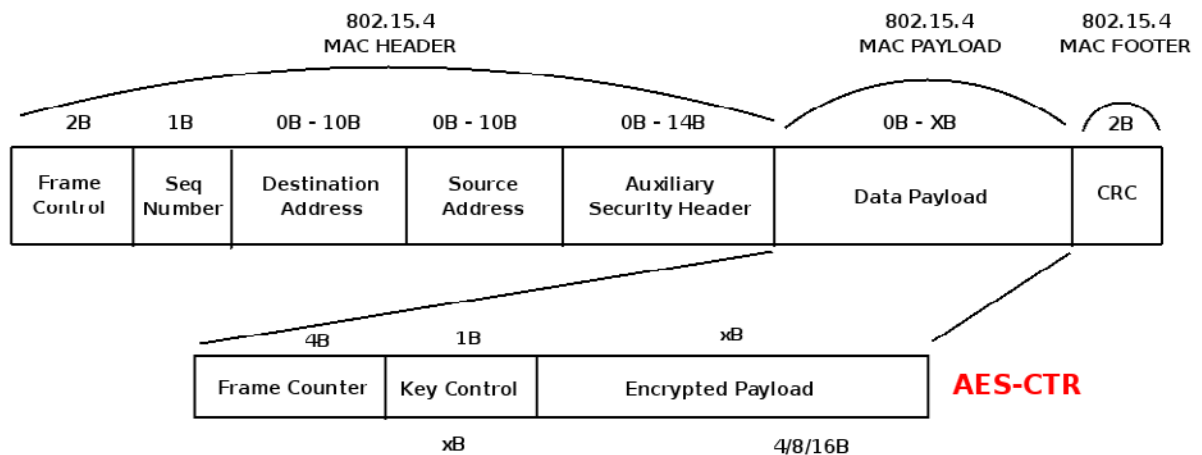


Figure 33: AES-CTR Encryption Frame

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. They 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.

14.2. 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.

14.2.1. Encryption Enable

Enables the 128bits AES encryption in the modules.

Example of use:

```
{
  xbeeZB.encryptionMode(1); // Enable encryption mode
}
```

Related Variables

xbeeZB.encryptMode → stores if security is enabled or not

14.2.2. Encryption Options

Configures options for encryption. Available values are:

- 0x01 : Send the security key unsecured over the air during joins
- 0x02 : Use Trust Center

Example of use:

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

Related Variables

xbeeZB.encryptOptions → stores the options for security

14.2.3. 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

14.2.4. 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="WasmoteNetwKey!";
  xbeeZB.setNetworkKey(KEY); // Set Link Key
}
```

Related Variables

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

14.2.5. 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
}
```

14.3. 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.

14.3.1. 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.

14.3.2. 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.

14.4. Examples

14.4.1. Configuring a network using security without a Trust Center

This is the easiest way to cypher the communications in a PAN. It is needed to set a valid 'Network Key' and the 'Link Key' in the coordinator and the rest of the nodes to zero. In this case, when nodes join the network, coordinator will send 'Network Key' unencrypted and they will store this key to communicate inside the network.

Coordinator

```
{
  char* NETKEY="WaspMoteNetwKey!";
  char* LINKKEY="";
  xbeeZB.encryptionMode(1); // Enable Encryption
  xbeeZB.setLinkKey(NETKEY); // Set Network Key
  xbeeZB.setNetworkKey(LINKKEY); // Set Link Key
}
```

Joining Devices

```
{
  char* LINKKEY="";
  xbeeZB.encryptionMode (1); // Enable Encryption
  xbeeZB.setNetworkKey(LINKKEY); // Set Link Key
}
```

14.4.2. Configuring a network using security and the coordinator as a Trust Center

This is the safest way to secure a network. In this case, 'Network Key' unsecured sending is avoided and a secure communication inside the network is assured. When a node is joining the network, the 'Network Key' is sent by the Trust Center encrypted with the 'Link Key'.

Coordinator: 'Network Key' and 'Link Key' must be set to the selected ones. 'Encryption Options' must be set to enable Trust Center.

Joining Devices: 'Link Key' must be the same as in coordinator and Trust Center must be enabled in 'Encryption Options'.

Coordinator

```
{
  char* NETKEY="WaspMoteNetwKey!";
  char* LINKKEY="WaspMoteLinkKey!";
  xbeeZB.encryptionMode (1); // Enable Encryption
  xbeeZB.setEncryptionOptions(0x02); // Enable Trust Center
  xbeeZB.setLinkKey(NETKEY); // Set Network Key
  xbeeZB.setNetworkKey(LINKKEY); // Set Link Key
}
```

Joining Devices

```
{
  char* LINKKEY="WaspMoteLinkKey!"
  xbeeZB.encryptionMode (1); // Enable Encryption
  xbeeZB.setEncryptionOptions(0x02); // Enable Trust Center
  xbeeZB.setNetworkKey(LINKKEY); // Set Link Key
}
```

15. Code examples and extended information

For more information about the WaspMote hardware platform go to:

<http://www.libelium.com/waspmote>

<http://www.libelium.com/support/waspmote>

<http://www.libelium.com/development/waspmote>