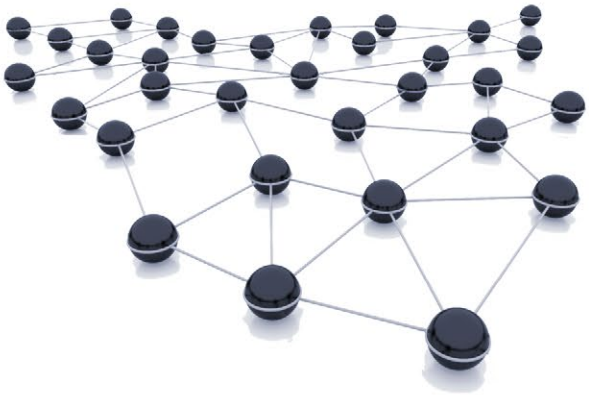
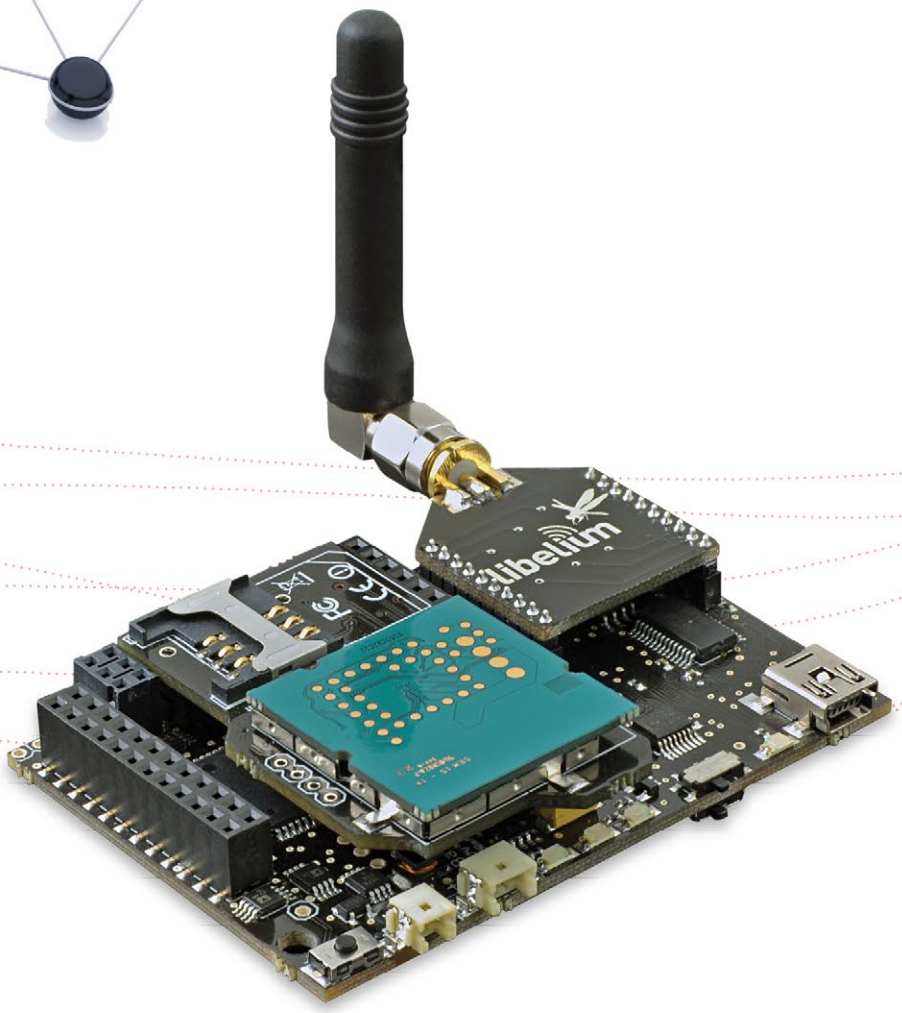
Waspmote 900MHz

Networking Guide





libelium

wasp mote

Document Version: v4.8 - 01/2016

INDEX

1. Hardware 4

2. General Considerations 6
2.1. Waspmote Libraries 6
2.1.1. Waspmote XBee Files 6
2.1.2. Constructor 6
2.2. API Functions 6
2.3. API extension 7
2.4. Waspmote reboot 7
2.5. Constants pre-defined 7

3. Initialization 8
3.1. Setting ON 8
3.2. Setting OFF 10

4. Node Parameters 11
4.1. MAC Address 11
4.2. PAN ID 12
4.3. Node Identifier 12
4.4. Channel 13

5. Power Gain and Sensibility 14
5.1. Received Signal Strength Indicator 14

6. Networking methods 15
6.1. Topologies 15
6.2. Addressing 15
6.3. Maximum payloads 16
6.4. Sending Data 16
6.4.1. Using Waspmote Frame 16
6.4.2. Sending function 16
6.4.3. Examples 17
6.5. Receiving Data 18
6.5.1. Receiving function 18
6.5.2. Examples 18

7. Node Discovery 19
7.1. Structure used in Discovery 19
7.2. Searching specific nodes 20
7.3. Node discovery to a specific node 20
7.4. Node Discovery Time 20

8. Sleep Options 21
8.1. Sleep Mode 21
8.2. Asynchronous Pin Sleep Mode 21
8.3. ON/OFF Modes 21

9. Security and Data Encryption 22
9.1. XBee 900 Security and Data encryption Overview 22
9.2. Security in API libraries 22
9.2.1. Encryption Enable 22
9.2.2. Encryption Key 22
9.3. Security in a network 23

10. Code examples and extended information 24

11. API changelog 25

12. Documentation changelog 26

1. Hardware

Module	Frequency	Tx Power	Sensitivity	Channels	Distance
XBee 900	902-928MHz	50mW	-100dBm	12	10km

Figure : XBee 900MHz

Note: The XBee 900 MHz module is provided with 4.5dBi antenna, which enables maximum range. The only exception is Smart Parking; in this case the antenna is smaller, 0dBi, to fit inside the enclosure.

The frequency used is the 900MHz band, using 12 channels with a bandwidth of **2.16MHz** per channel and a transmission rate of 156.25kbps. The use of this module is only allowed in the United States, Canada, and other few countries. More information can be obtained about the **Certifications** in Waspmote Technical Guide.

Note: Due to the propagation characteristics of the 868/900 MHz band, the near field effect could make that 2 modules cannot communicate if they are placed very close (< 1 m). We suggest to keep a minimum distance of 3 or 4 meters between modules.

Note: It is not recommended to work without an antenna screwed to the module. The module could be damaged due to RF reflections.

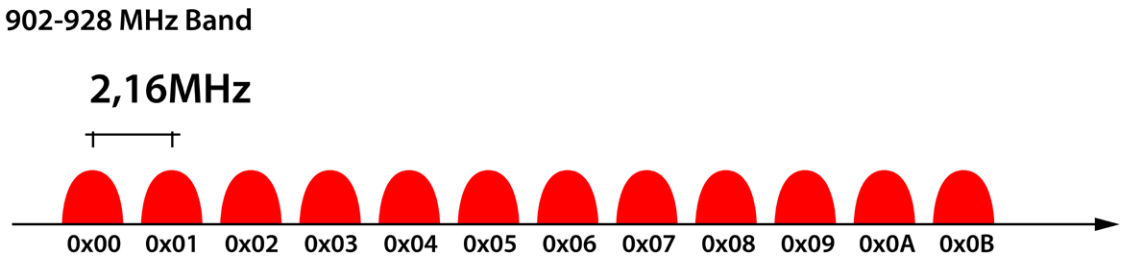


Figure : Channel frequencies in the 900MHz band

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

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

```
{
  xbee900.setEncryptionMode(1);
  xbee900.setLinkKey(key);
}
```

The classic topology for this type of network is Star topology, as the nodes can establish point to point connections with brother nodes through the MAC address.

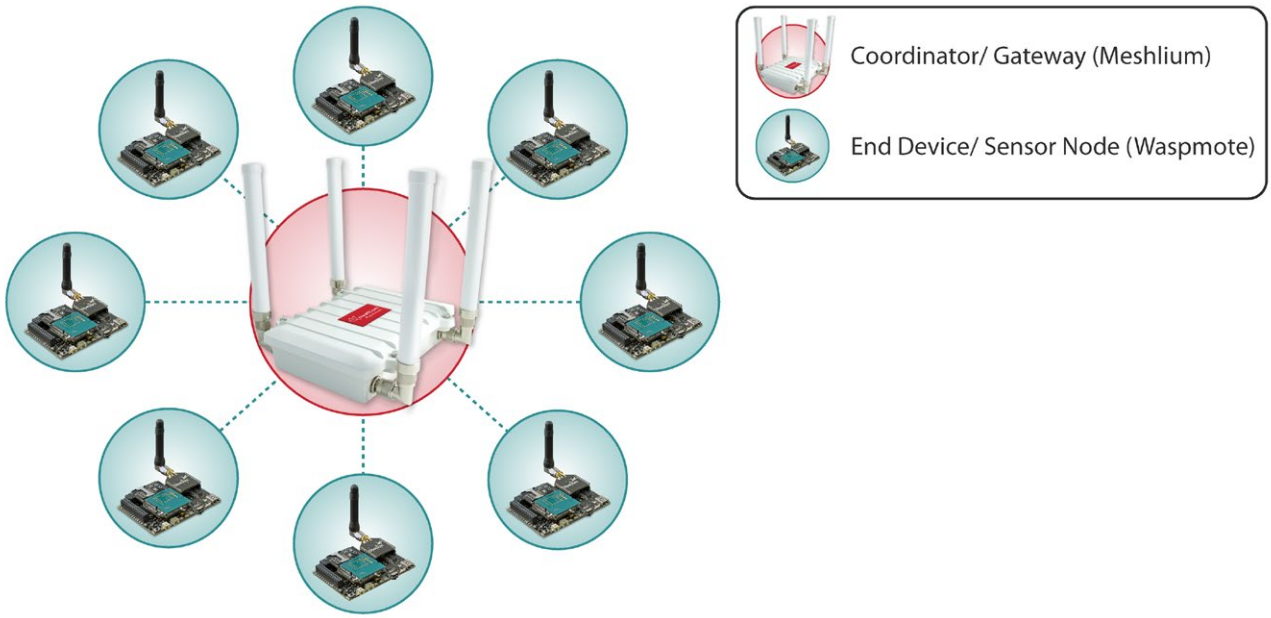


Figure : Star topology

2. General Considerations

2.1. Waspmote Libraries

2.1.1. Waspmote XBee Files

WaspXBeeCore.h, WaspXBeeCore.cpp, WaspXBee900.h, WaspXBee900.cpp

It is mandatory to include the XBee900 library when using this module. The following line must be introduced at the beginning of the code:

```
#include <WaspXBee900.h>
```

2.1.2. Constructor

To start using the Waspmote XBee 900MHz library, an object from class 'WaspXBee900' must be created. This object, called `xbee900`, is created inside the Waspmote XBee 900 library and it is public to all libraries. It is used through the guide to show how the Waspmote XBee 900MHz library works.

When creating this constructor, no 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

```
{
  xbee900.getOwnMacLow(); // Get 32 lower bits of MAC Address
  xbee900.getOwnMacHigh(); // Get 32 upper bits of MAC Address
}
```

Related Variables

`xbee900.sourceMacHigh[0-3]` → stores the 32 upper bits of MAC address
`xbee900.sourceMacLow [0-3]` → stores the 32 lower bits of MAC address

When returning from 'xbee900.getOwnMacLow' the exposed variable 'xbee900.sourceMacLow' will be filled with the appropriate values. Before calling the function, the exposed variable is created but it is empty.

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

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

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

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

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

Example of use

```
{
  xbee900.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

```
{
   xbee900.sendCommandAT("CH#"); // Executes command ATCH
}
```

Related Variables

`xbee900.commandAT[0-100]` → stores the response given by the module up to 100 bytes

• Sending AT commands example:

http://www.libelium.com/development/waspmote/examples/900-11-send-atcommand

2.4. Waspmote reboot

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

2.5. Constants pre-defined

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

- `MAX_DATA`: (default value is 300) it defines the maximum available data size for a packet. This constant must be equal or bigger than the data is sent on each packet. This size shouldn't be bigger than 1500.
- `MAX_PARSE`: (default value is 300) it defines the maximum data that is parsed in each call to `treatData()`. If more data are received, they will be stored in the UART buffer until the next call to `treatData()`. However, if the UART buffer is full, the following data will be written on the buffer, so be careful with this matter.
- `MAX_BROTHERS`: (default value is 5) it defines the maximum number of brothers that can be stored.

3. Initialization

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

3.1. Setting ON

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

It returns nothing.

The initialized variables are:

- **protocol** : specifies the protocol used (XBEE 900 in this case).
- **pos** : specifies the position to use in received packets.
- **discoveryOptions** : specifies the options in Node Discovery.
- **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.
- **timeRSSI** : specifies the time RSSI LEDs are on.

Example of use

```
{
   xbee900.ON();
}
```

Expansion Radio Board (XBee 900)

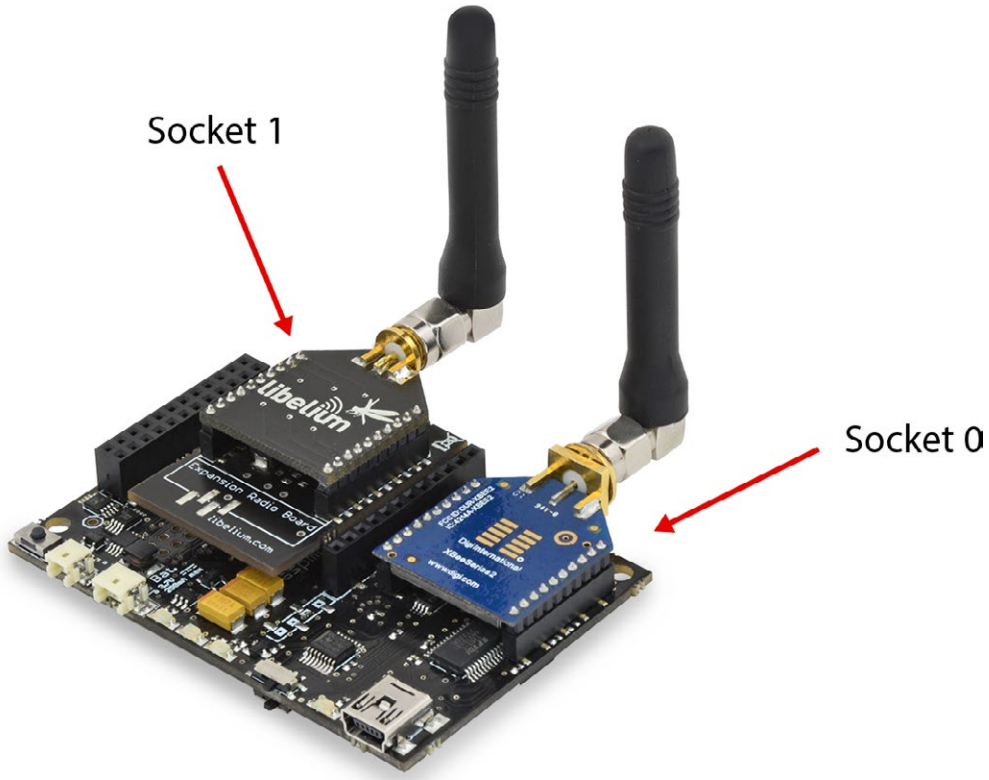
The Expansion Board allows to connect two communication modules at the same time in the Waspmote sensor platform. This means a lot of different combinations are possible using any of the wireless radios available for Waspmote: 802.15.4, ZigBee, DigiMesh, 868 MHz, 900 MHz, LoRaWAN, LoRa, Sigfox, Bluetooth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

Some of the possible combinations are:

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

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

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



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

Selecting SOCKET0 (both are valid):

```
xbee900.ON();
xbee900.ON(SOCKET0);
```

Selecting SOCKET1:

```
xbee900.ON(SOCKET1);
```

In the case two XBee-900 modules are needed (each one in each socket), it will be necessary to create a new object from WaspXBee900 class. By default, there is already an object called 'xbee900' normally used for regular SOCKET0.

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

```
WaspXBee900 xbee900_2 = WaspXBee900();
```

Finally, it is necessary to initialize both modules. For example, xbee900 is initialized in SOCKET0 and xbee900_2 in SOCKET1 as follows:

```
xbee900.ON(SOCKET0);
xbee900_2.ON(SOCKET1);
```

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

WARNING:

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

3.2. Setting OFF

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

Example of use

```
{
   xbee900.OFF();
}
```

4. Node Parameters

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

4.1. MAC Address

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

It identifies uniquely a node inside a network due to it can not be modified and it is given by the manufacturer.

Example of use

```
{
  xbee900.getOwnMacLow(); // Get 32 lower bits of MAC Address
  xbee900.getOwnMacHigh(); // Get 32 upper bits of MAC Address
}
```

Related Variables

`xbee900.sourceMacHigh[0-3]` → stores the 32 upper bits of MAC address
`xbee900.sourceMacLow [0-3]` → stores the 32 lower bits of MAC address

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

Figure : MAC address

4.2. PAN ID

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

Example of use

```
{
   uint8_t panid[2]={0x33,0x31};
   xbee900.setPAN(panid);
   xbee900.getPAN();
}
```

Related Variables

`xbee900.PAN_ID[0-1]` → stores the 16-bit PAN ID.

• XBee configuration example:

http://www.libelium.com/development/waspmote/examples/900-01-configure-xbee-parameters

4.3. Node Identifier

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

Example of use

```
{
   xbee900.setNodeIdentifier(“node01”);
   xbee900.getNodeIdentifier();
}
```

Related Variables

`xbee900.nodeID[0-19]` → stores the 20-byte max string Node Identifier

4.4. Channel

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

XBee 900Mhz modules define 12 channels (5 for the international version):

- 902-928MHz : 12 channels

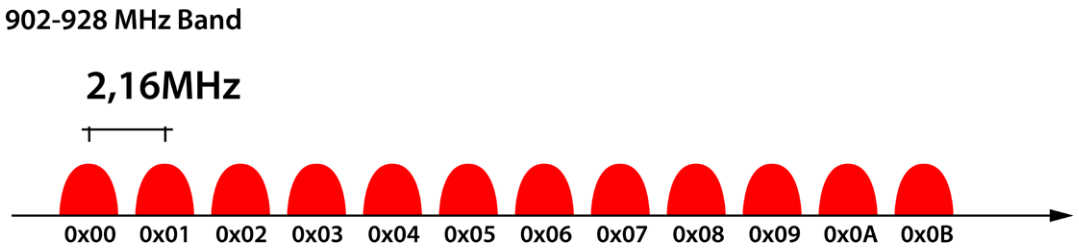


Figure : Operating Frequency Bands on 900MHz

Channel Number	Frequency
0x00 – Channel 0	902 – 904,16 MHz
0x01 – Channel 1	904,16 – 906,32 MHz
0x02 – Channel 2	906,32 – 908,48 MHz
0x03 – Channel 3	908,48 – 910,64 MHz
0x04 – Channel 4	910,64 – 912,80 MHz
0x05 – Channel 5	912,80 – 914,96 MHz
0x06 – Channel 6	914,96 – 917,12 MHz
0x07 – Channel 7	917,12 – 919,28 MHz
0x08 – Channel 8	919,28 – 921,44 MHz
0x09 – Channel 9	921,44 – 923,6 MHz
0x0A – Channel 10	923,6 – 925,76 MHz
0x0B – Channel 11	925,76 – 928 MHz

Figure : Channel Frequency Numbers on 900MHz

Note: The module manages automatically the band assignation with a number of hopping frequency pattern.

Example of use

```
{
  xbee900.setChannel(0x0B);
  xbee900.getChannel();
}
```

Related Variables

`xbee900.channel` → stores the operating channel

• XBee configuration example:

http://www.libelium.com/development/waspmote/examples/900-01-configure-xbee-parameters

5. Power Gain and Sensibility

When configuring a node and a network, one important parameter is related with RSSI.

5.1. 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:

```
{
  xbee900.getRSSI();
}
```

Related Variables

`xbeeDM.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.

• Get RSSI example:

http://www.libelium.com/development/waspmote/examples/900-05-get-rssi

6. Networking methods

Note: *It is important to keep in mind that XBee networks are defined by the networking parameters. Every XBee module within a network must share the same networking parameters. In the case of the XBee 900, every node in a network must have the same:*

- *PAN ID*
- *Encryption configuration*

6.1. Topologies

XBee 900 provides a star topology to create a network:

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

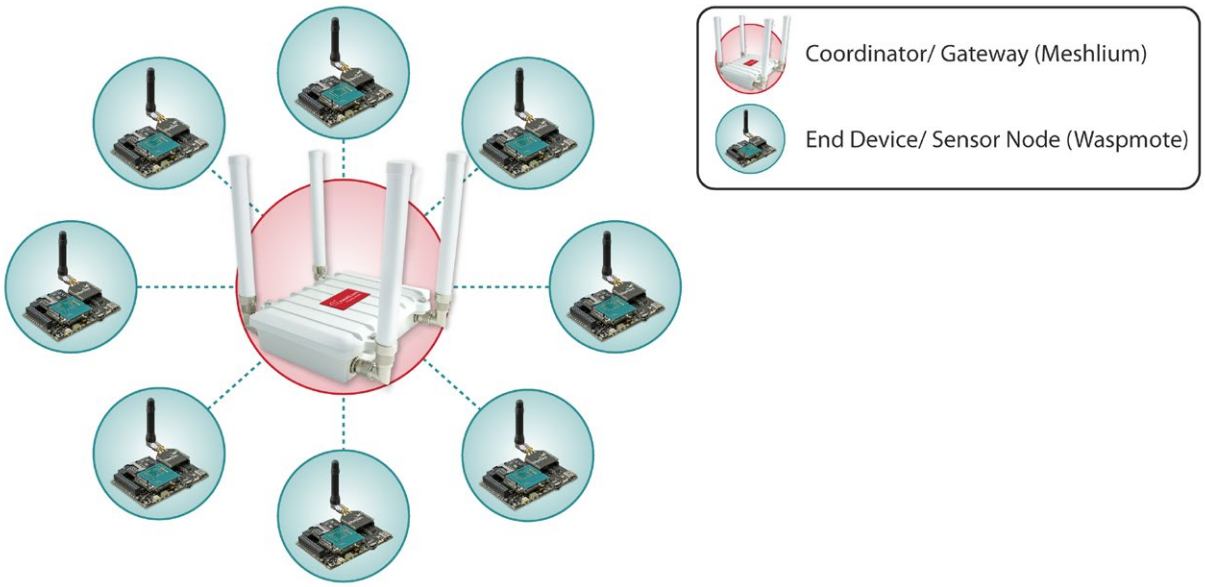


Figure : Star Topology

6.2. Addressing

Every RF data packet sent over-the-air contains a Source Address and Destination Address field in its header. XBEE 900 supports long 64-bit addresses. A unique 64-bit IEEE source address is assigned at the factory.

XBee 900 supports unicast and broadcast transmissions:

- Unicast: The unicast mode is the only mode that supports acknowledgements (ACKs). When a packet is sent using unicast mode, the receiving module sends an affirmative response to the sending module. If the sending module does not receive the ACK, it will re-send the packet up to ten times or until the ACK is received. To send a unicast message, the 64-bit receiver's address must be set correctly.
- Broadcast: Used to transmit all modules in the same network. Any RF module within range will accept a packet that contains a broadcast address. When a packet is sent using broadcast mode, receiving modules do not send ACKs. All broadcast packets are automatically transmitted four times to ensure it is received. To send a broadcast message, the receiver address must be set to 0x000000000000FFFF.

6.3. Maximum payloads

Depending on the addressing mode, a maximum data payload is defined:

	Unicast	Broadcast
Encrypted	80Bytes	80Bytes
Un-Encrypted	100Bytes	100Bytes

Figure : Maximum Payloads Size

6.4. Sending Data

6.4.1. Using Waspmote Frame

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

http://www.libelium.com/development/waspmote/documentation/data-frame-guide/

6.4.2. Sending function

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

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

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

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

- Define broadcast mode:

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

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

- Send strings:

```
{
   char data[] = "this_is_the_data_field";
   xbee900.send( rx_address, data);
}
```

- Send Waspmote Frames:

```
{
   xbee900.send( rx_address, frame.buffer, frame.length );
}
```

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

```
{
   uint8_t  data[5] = {0x00, 0x01, 0x54, 0x76, 0x23};
   xbee900.send( rx_address, data, 5);
}
```

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

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

Parameter range: From 0 to 10
Default: 3

Example of use:

```
{
   xbee900.setSendingRetries(10);
}
```

Related variables:

`xbee900._send_retries` → stores the maximum number of application-level retries

6.4.3. Examples

• Send packets in unicast mode:

http://www.libelium.com/development/waspmote/examples/900-02-send-packets

• Send packets in broadcast mode:

http://www.libelium.com/development/waspmote/examples/900-04a-send-broadcast

• Send packets using the expansion board:

http://www.libelium.com/development/waspmote/examples/900-06a-expansion-board-send

• Complete example, send packets in unicast mode and wait for a response:

http://www.libelium.com/development/waspmote/examples/900-08a-complete-example-send

6.5. Receiving Data

6.5.1. Receiving function

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

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

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

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

'0' ⟶ OK: The command has been executed with no errors
'1' ⟶ Error: timeout when receiving answer
'2' ⟶ Error: Frame Type is not valid
'3' ⟶ Error: Checksum byte is not available
'4' ⟶ Error: Checksum is not correct
'5' ⟶ Error: Error escaping character in checksum byte
'6' ⟶ Error: Error escaping character within payload bytes
'7' ⟶ Error: Buffer full. not enough memory space

Example of use:

```
{
   uint8_t  error;
   error = xbee900.receivePacketTimeout( 10000 );
}
```

Related variables:

`xbee900._payload[]` ⟶ Buffer where the received packet is stored
`xbee900._length` ⟶ Length of the buffer
`xbee900._srcMAC[0-7]` ⟶ Source's MAC address

6.5.2. Examples

- Receiving packets example:
  http://www.libelium.com/development/waspmote/examples/900-03-receive-packets
- Receive packets in broadcast mode (the same procedure as if it was unicast mode):
  http://www.libelium.com/development/waspmote/examples/900-04b-receive-broadcast
- Receive packets using the expansion board:
  http://www.libelium.com/development/waspmote/examples/900-06b-expansion-board-reception
- Complete example, receive packets and send a response back to the sender:
  http://www.libelium.com/development/waspmote/examples/900-08b-complete-example-receive

7. Node Discovery

XBee modules provide some features for discovering and searching nodes.

7.1. Structure used in Discovery

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

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

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

- `MY`: 16-bit Network Address of the reported module (not relevant in this protocol).
- `SH[4]` and `SL[4]`: 64-bit MAC Source Address of the reported module.
- `NI`: Node Identifier of the reported module
- `PMY`: Parent 16-bit network address. It specifies the 16-bit network address of its parent.
- `DT`: Device Type. It specifies if the node is a Coordinator, Router or End Device.
- `ST`: Status. Reserved by XBee 900.
- `PID`: Profile ID. Profile ID used to application layer addressing.
- `MID`: Manufacturer ID. ID set by the manufacturer to identify the module.
- `RSSI`: Not returned in XBee 900.

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

Example of use:

```
{
   xbee900.scanNetwork();
}
```

Related variables:

`xbee900.totalScannedBrothers` ⟶ stores the number of discovered brothers
`xbee900.scannedBrothers` ⟶ Node structure array that stores the info

- Scan network example:
  http://www.libelium.com/development/waspmote/examples/900-09-scan-network

7.2. Searching specific nodes

Another possibility for discovering a node is searching for a specific one. This search is based on using the Node Identifier. The NI of the node to discover is used as the input in the API function responsible of this purpose.

Example of use:

```
{
   uint8_t  mac[8];
   xbee900.nodeSearch("node01", mac);
}
```

Related variables:

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

• Node search example:

http://www.libelium.com/development/waspmote/examples/900-10a-node-search-tx
http://www.libelium.com/development/waspmote/examples/900-10b-node-search-rx

7.3. Node discovery to a specific node

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

Example of use:

```
{
   xbee900.scanNetwork("node01");
}
```

Related variables:

`xbee900.totalScannedBrothers` ⟶ stores the number of discovered brothers. Must be '1'.
`xbee900.scannedBrothers` ⟶ Node structure array that stores the info

7.4. Node Discovery Time

It is the amount of time a node will wait for responses from other nodes when performing a ND. Range: 0X20 to 0X2EE0 [x100ms]. Default value: 0x0082.

Example of use:

```
{
   uint8_t time[2]={0x00,0x82};
   xbee900.setScanningTime(time);
   xbee900.getScanningTime();
}
```

Available Information:

`xbee900.scanTime[0-1]`⟶ stores the time a node will wait for responses.

8. Sleep Options

Sleep Modes enable the XBee module to enter into low-power consumption states when they are not in use.

8.1. Sleep Mode

The sleep modes are enabled with the following API function. The available values are:

0: Normal mode. In this mode, a node will not sleep.
1: Asynchronous Pin Sleep Mode . The module can enter a low-power consumption state according to the value of the sleep pin.

Example of use:

```
{
  xbee900.setSleepMode(0); // Set Normal Mode
  xbee900.setSleepMode(1); // Set Sleep Mode to Pin Sleep
  xbee900.getSleepMode(); // Get the Sleep Mode used
}
```

Related Variables

`xbee900.sleepMode` → stores the sleep mode in a module

• Sleep mode example:

http://www.libelium.com/development/waspmote/examples/900-07-set-low-power-mode

8.2. Asynchronous Pin Sleep Mode

Pin sleep allows the module to sleep and wake according to the state of the sleep pin. Pin sleep mode is enabled by setting the Sleep Mode to 1. When Sleep_RQ is asserted (high), the module will finish any transmit or receive operations and enter a low-power state. The module will wake from pin sleep when the Sleep_RQ pin is de-asserted (low).

Example of use:

```
{
 xbee900.setSleepMode(1); // Set Sleep Mode to Asynchronous Pin Sleep Mode
 xbee900.sleep(); // Set XBee to sleep

 delay(5000); // wait 5 seconds
 xbee900.wake(); // Wake up the XBee
}
```

8.3. ON/OFF Modes

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

Example of use:

```
{
  xbee900.setMode(XBEE_ON); // Powers XBee up
  xbee900.setMode(XBEE_OFF); // Powers XBee down
}
```

9. Security and Data Encryption

9.1. XBee 900 Security and Data encryption Overview

The encryption algorithm used in XBee 900 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.

9.2. Security in API libraries

As explained previously, XBee 900 provides secure communications inside a network using 128-bit AES encryption. The API functions enable using security and data encryption.

9.2.1. Encryption Enable

Enables the 128-bit AES encryption in the modules.

Example of use:

```
{
  xbee900.setEncryptionMode(0); // Disable encryption mode
  xbee900.setEncryptionMode(1); // Enable encryption mode
  xbee900.getEncryptionMode(); // Get encryption mode
}
```

Related Variables

`xbee900.encryptMode` → stores if security is enabled or not

• XBee configuration example:

http://www.libelium.com/development/waspmote/examples/900-01-configure-xbee-parameters

9.2.2. Encryption Key

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

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

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

Example of use

```
{
  char KEY[] = ”WaspmoteLinkKey!” ;
  xbee900.setLinkKey(KEY);
}
```

Related Variables

`xbee900.linkKey` → stores the key that has been set in the network

- XBee configuration example:
  http://www.libelium.com/development/waspmote/examples/900-01-configure-xbee-parameters

9.3. Security in a network

When creating or joining a network, using security is highly recommended to prevent the network from attacks or intruder nodes.
It is necessary to enable security and set the same encryption key in all nodes in order to set security in a network. If not, it won't be possible to communicate between different XBee modules.

10. Code examples and extended information

In the Waspmote Development section you can find complete examples:

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

Example:

```
#include <WaspXBee900.h>
#include <WaspFrame.h>

// Destination MAC address
////////////////////////////////////////
char RX_ADDRESS[] = “0013A2004066EF95”;
////////////////////////////////////////

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

// define variable
uint8_t error;

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

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

    // init XBee
  xbee900.ON();
}

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

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

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

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

  // send XBee packet
  error = xbee900.send( RX_ADDRESS, frame.buffer, frame.length );

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

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

11. API changelog

Keep track of the software changes on this link:

www.libelium.com/development/waspmote/documentation/changelog/#900

12. Documentation changelog

From v4.7 to v4.8

- References to the new LoRaWAN module

From v4.6 to v4.7

- References to the new Sigfox module

From v4.5 to v4.6

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

From v4.4 to v4.5

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

From v4.3 to v4.4

- Expansion Radio Board section updated
- Some advice about RF performance

From v4.2 to v4.3

- Explanation about the old 0dBi antenna: only available for Smart Parking now

From v4.1 to v4.2

- API changelog updated to API v006

From v4.0 to v4.1

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