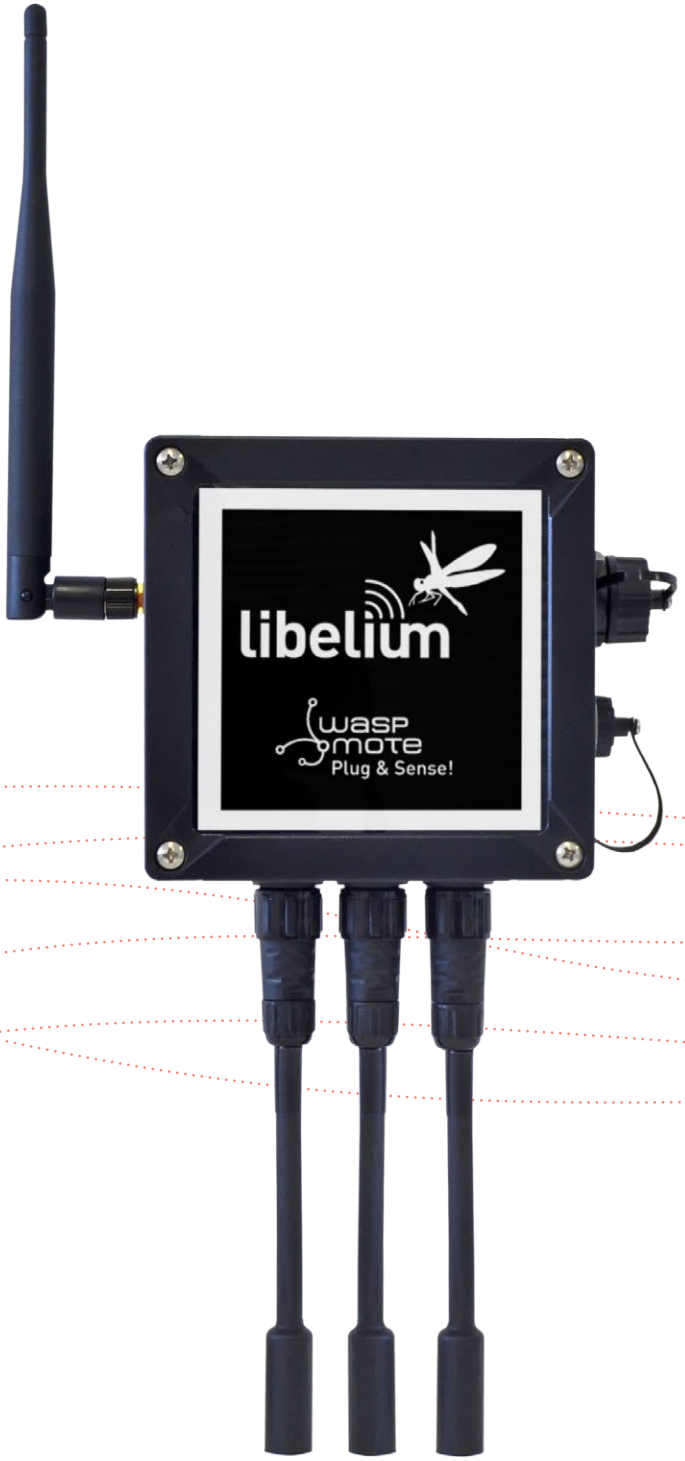
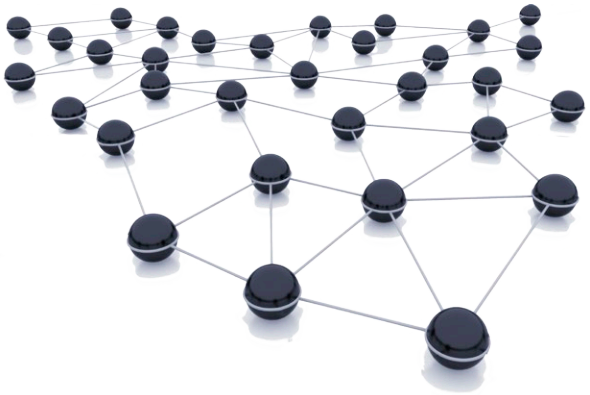
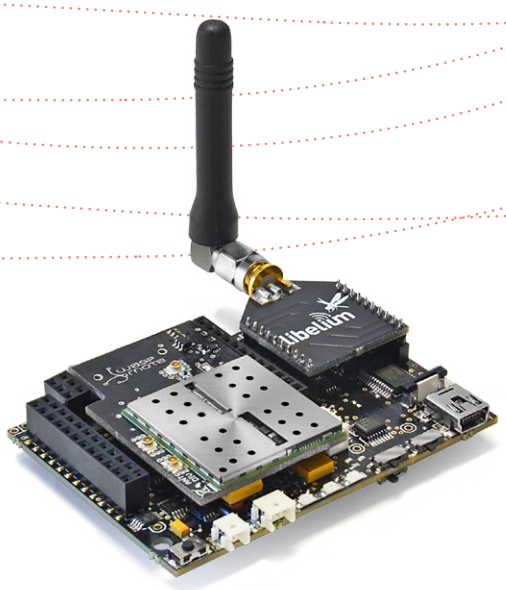
Waspmote Digimesh

Networking Guide







libelium

wasp mote

Document version: v7.0 - 02/2017

INDEX

1. Introduction

This guide explains the XBee DigiMesh features and functions. There are no great variations in this library for our new product lines Waspmote v15 and Plug & Sense! v15, released on October 2016.

Anyway, if you are using previous versions of our products, please use the corresponding guides, available on our Development website.

You can get more information about the generation change on the document "New generation of Libelium product lines".

2. Hardware

The XBee-802.15.4 modules can use an optional firmware (DigiMesh) so that they are able to create mesh networks instead of the usual point to point topology. This firmware has been developed by Digi to allow modules to sleep, synchronize themselves and work on equal terms, avoiding the use of router nodes or coordinators that have to be permanently powered on. Characteristics of the implemented protocol:

- **Self Healing:** any node can join or leave the network at any moment.
- **All nodes are equal:** There are no father-son relationships.
- **Silent protocol:** reduced routing heading due to using a reactive protocol similar to AODV (Ad hoc On-Demand Vector Routing).
- **Route discovery:** instead of keeping a route map, routes are discovered when they are needed.
- **Selective ACKs:** only the recipient responds to route messages.
- **Reliability:** the use of ACKs ensures data transmission reliability.
- **Sleep Modes:** low energy consumption modes with synchronization to wake at the same time.

The classic topology of this type of network is mesh, as the nodes can establish point to point connections with brother nodes through the use of parameters such as the MAC or network address or by making **multi-jump connections.**

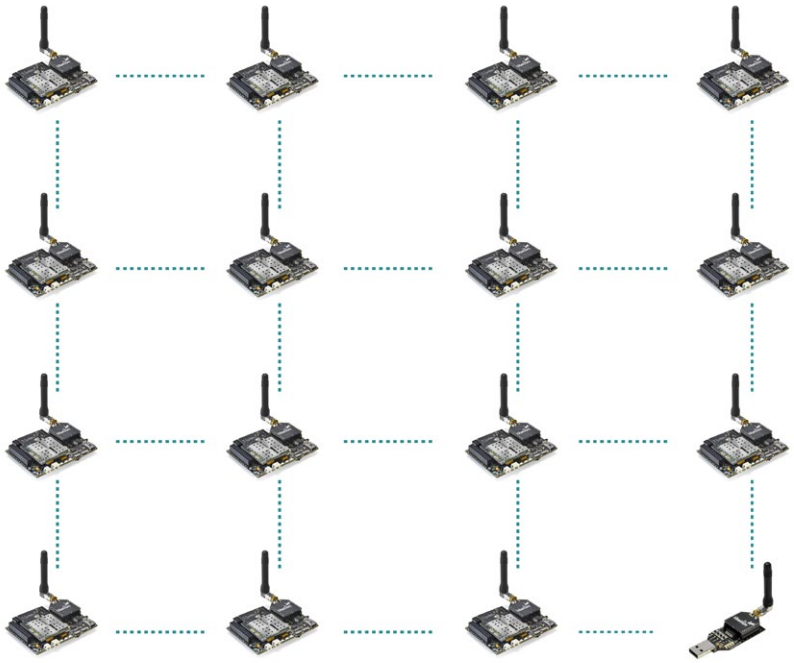


Figure : Mesh topology

Module	Frequency	Tx Power	Sensitivity	Channels	Range*
XBee-PRO DigiMesh	2.405 – 2.465 GHz	18 dBm	-100 dBm	12	1500 m

** To determine your range, perform a range test under your operating conditions*

The XBee-PRO DigiMesh modules use the same hardware as the XBee-PRO 802.15.4. So it is possible to use one protocol or another by changing the firmware. The user must keep in mind that depending on the regulations applied the transmission power should be set to the correct value. For instance, ETSI defines a maximum level of 10 dBm transmit power output.

The process of changing the firmware can be done with a Gateway and Digi's X-CTU program. See our X-CTU tutorials:

http://www.libelium.com/es/development/waspmote/documentation/changing-the-xbee-firmware-from-802-15-4-to-digimesh/

The XBee DigiMesh modules are based on the standard **IEEE 802.15.4** that supports functionalities enabling mesh topology use.

3. General considerations

3.1. Waspmote libraries

3.1.1. Waspmote XBee files

WaspXBeeCore.h, WaspXBeeCore.cpp, WaspXBeeDM.h, WaspXBeeDM.cpp

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

```
#include <WaspXBeeDM.h>
```

3.1.2. Constructor

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

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

3.2. 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:

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

Related variables:

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

When returning from 'xbeeDM.getOwnMacLow' the related variable 'xbeeDM.sourceMacLow' will be filled with the appropriate values. Before calling the function, the related 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 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:

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

3.3. 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 using the `sendCommandAT()` function.

Example of use:

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

Related variables:

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

Send AT commands example:
http://www.libelium.com/development/waspmote/examples/dm-12-send-at-command

3.4. Waspmote reboot

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

3.5. Constants 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 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`: (default value is 5) it defines the maximum number of brothers that can be stored.

4. 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.

4.1. 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 (115200 bps by default).

It returns nothing.

The initialized variables are:

- **protocol:** specifies the protocol used (DIGIMESH in this case).
- **pos**: specifies the position to use in received packets.
- **discoveryOptions**: specifies the options in Node Discovery.
- **awakeTime**: specifies the time to be awake before go sleeping.
- **sleepTime**: specifies the time to be sleeping.
- **scanChannels**: specifies the channels to scan.
- **scanTime**: specifies the time to scan each channel.
- **timeEnergyChannel**: specifies the time the channels will be scanned.
- **encryptMode**: specifies if encryption mode is enabled.
- **powerLevel**: specifies the power transmission level.
- **timeRSSI**: specifies the time RSSI LEDs are on.
- **sleepOptions**: specifies the options for sleeping.
- **networkHops**: specifies the maximum number of hops expected to be seen in a network route.
- **netDelaySlots**: specifies the maximum random number of network delay slots before rebroadcasting a network packet.
- **netRouteRequest**: specifies the maximum number of route discovery retries allowed to find a path to the destination node.
- **meshNetRetries**: specifies the maximum number of network packet delivery attempts.

Example of use:

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

Expansion Radio Board (XBee DigiMesh)

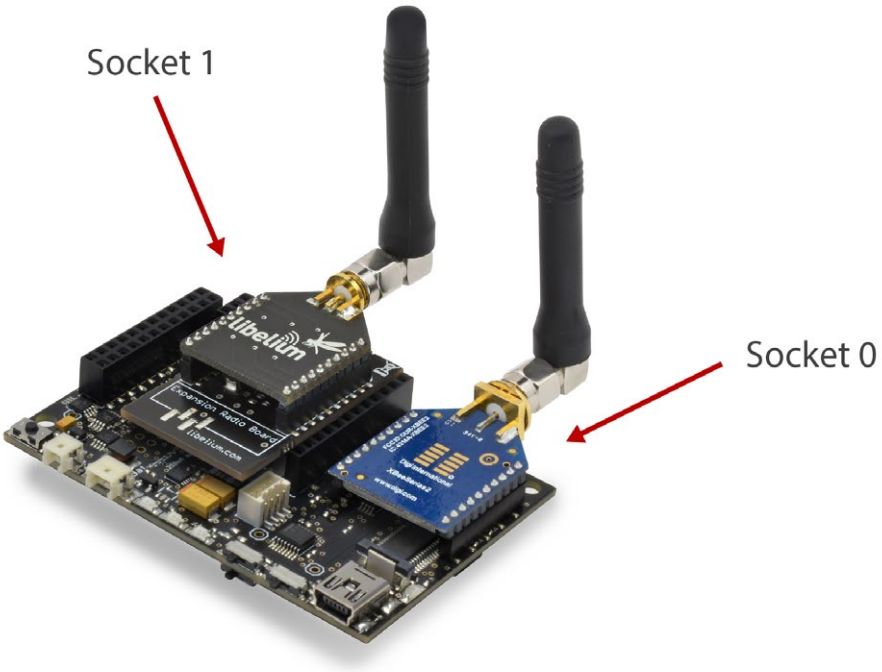
The Expansion Board allows to connect two communication modules at the same time in the Waspmote sensor platform. This means a lot of different combinations are possible using any of the wireless radios available for Waspmote: 802.15.4, ZigBee, DigiMesh, 868 MHz, 900 MHz, LoRa, WiFi, GPRS, GPRS+GPS, 3G, 4G, Sigfox, LoRaWAN, Bluetooth Pro, Bluetooth Low Energy and RFID/NFC. 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 - 4G
- RS-232 - RFID/NFC
- WiFi - 3G
- CAN bus - Bluetooth
- etc.

Remark: *GPRS, GPRS+GPS, 3G and 4G modules do not need the Expansion Board to be connected to Waspmote. They can be plugged directly in the socket1.*

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



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

Selecting SOCKET0 (both are valid):

```
xbeeDM.ON();
xbeeDM.ON(SOCKET0);
```

Selecting SOCKET1:

```
xbeeDM.ON(SOCKET1);
```

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

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

```
WaspXBeeDM xbeeDM_2 = WaspXBeeDM();
```

Finally, it is necessary to initialize both modules. For example, xbeeDM is initialized in SOCKET0 and xbeeDM_2 in SOCKET1 as follows:

```
xbeeDM.ON(SOCKET0);
xbeeDM_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:
  - Agriculture v30 and Agriculture PRO v30: Incompatible with Watermark and solar radiation sensors
  - Events v30: Incompatible with interruption shift register
  - Gases v30: DIGITAL6 is incompatible with CO2 (SOCKET_2) and DIGITAL7 is incompatible with NO2 (SOCKET_3)
  - Smart Water v30: DIGITAL7 incompatible with conductivity sensor
  - Smart Water Ions v30: Incompatible with ADC conversion (sensors cannot be read if the Expansion Board is in use)
  - Gases PRO v30: Incompatible with SOCKET_2 and SOCKET_3
  - Cities PRO v30: Incompatible with SOCKET_3. I2C bus can be used. No gas sensor can be used.

4.2. Setting off

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

Example of use:

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

5. 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.

5.1. 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:

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

Related variables:

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

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

Figure : MAC adress

5.2. 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}; // array containing the PAN ID
  xbeeDM.setPAN(panid); // Set PANID
  xbeeDM.getPAN(); // Get PANID
}
```

Related variables:

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

XBee configuration example:

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

5.3. Node identifier

It is an ASCII string of 20 characters 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:

```
{
   xbeeDM.setNodeIdentifier("node01");
   xbeeDM.getNodeIdentifier();
}
```

Related variables:

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

5.4. Channel

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

When working on 2.4 GHz band, DigiMesh defines 16 channels to be used, 12 of them are available in this hardware version:

- 2.405-2.465 GHz: 12 channels

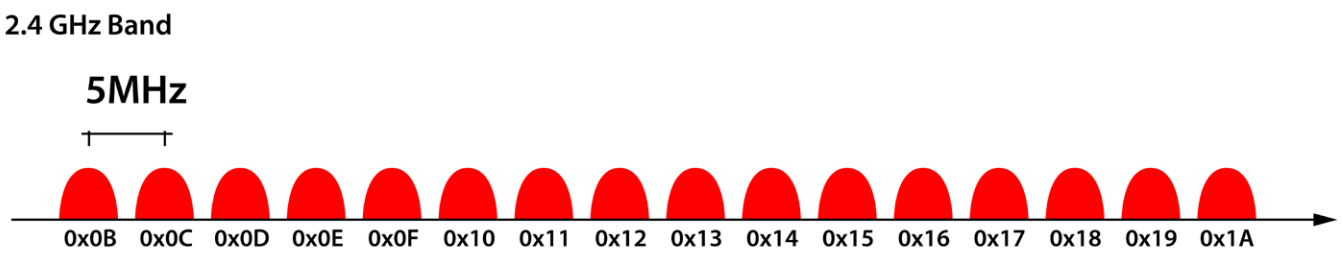


Figure : Operating frequency bands on 2.4 GHz

Channel Number	Frequency
0x0C – Channel 12	2.405 – 2.410 GHz
0x0D – Channel 13	2.410 – 2.415 GHz
0x0E – Channel 14	2.415 – 2.420 GHz
0x0F – Channel 15	2.420 – 2.425 GHz
0x10 – Channel 16	2.425 – 2.430 GHz
0x11 – Channel 17	2.430 – 2.435 GHz
0x12 – Channel 18	2.435 – 2.440 GHz
0x13 – Channel 19	2.440 – 2.445 GHz
0x14 – Channel 20	2.445 – 2.450 GHz
0x15 – Channel 21	2.450 – 2.455 GHz
0x16 – Channel 22	2.455 – 2.460 GHz
0x17 – Channel 23	2.460 – 2.465 GHz

Figure : Channels frequency numbers on 2.4 GHz

Example of use:

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

Related variables:

`xbeeDM.channel` → stores the operating channel

XBee configuration example:

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

5.5. Network hops

It specifies the (NH) maximum number of hops expected to be seen in a network route. This value does not limit the number of hops allowed, but it is used to calculate timeouts waiting for network acknowledgments. Parameter range: 1 to 0xFF. Default: 0x07.

Example of use:

```
{
  xbeeDM.setNetworkHops(0x07);
  xbeeDM.getNetworkHops();
}
```

Related variables:

`xbeeDM.networkHops` → stores the number of hops selected

5.6. Network delay slots

It specifies the (NN) maximum random number of network delay slots before rebroadcasting a network packet. One network delay slot is approximately 13 ms. Parameter range: 0 to 0x0A. Default: 0x03.

Example of use:

```
{
  xbeeDM.setNetworkDelaySlots(0x03);
  xbeeDM.getNetworkDelaySlots();
}
```

Related variables:

`xbeeDM.netDelaySlots` → stores the number of delay slots

5.7. Mesh network retries

It specifies the (MR) maximum number of network packet delivery attempts. If MR is non-zero, packets sent will request a network acknowledgment, and can be resent up to MR+1 times if no acknowledgments are received. Parameter range: 0 to 7. Default: 1.

Example of use:

```
{
  xbeeDM.setMeshNetworkRetries(0x01);
  xbeeDM.getMeshNetworkRetries();
}
```

Related Variables

`xbeeDM.meshNetRetries` → stores the number of network packet delivery attempts

6. Power gain and sensitivity

When configuring a node and a network, there is an important parameter related to power gain and sensitivity.

6.1. Power level

Power level (dBm) at which the RF module transmits conducted power.

The possible values are:

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

Figure : Power output level

Note: *dBm is a standard unit to measure power level taking as reference a 1mW signal.*

Values expressed in dBm can be easily converted to mW using the next formula:

mW = 10^(value dBm/10)

Graphics about transmission power are exposed next:

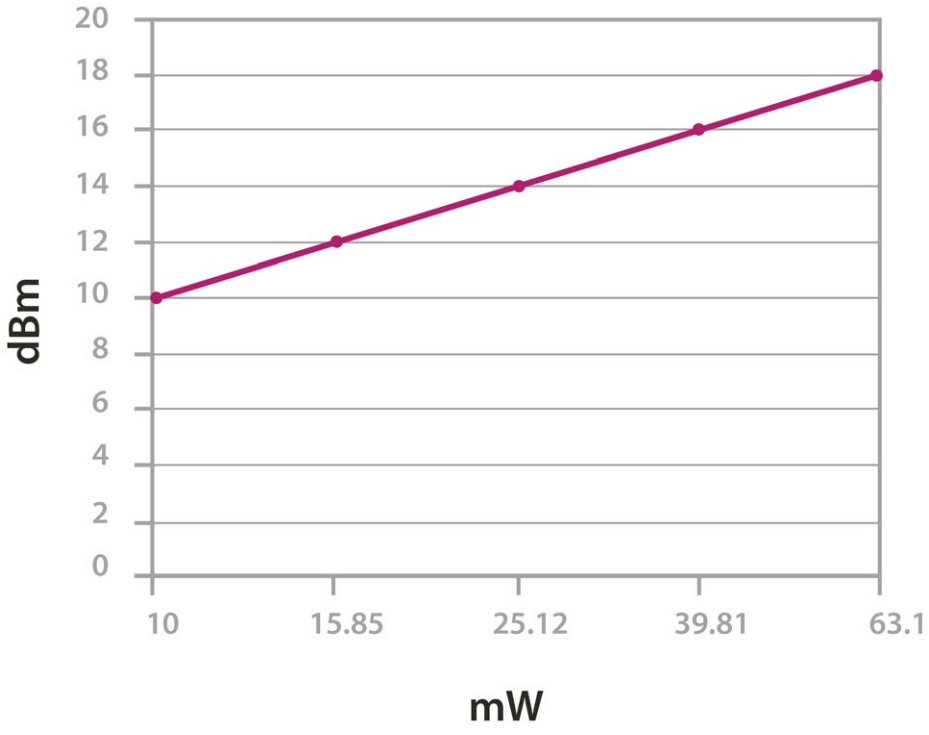


Figure : XBee-PRO DigiMesh output power level

Example of use:

```
{
  xbeeDM.setPowerLevel(0);
  xbeeDM.getPowerLevel();
}
```

Related variables:

`xbeeDM.powerLevel` → stores the power output level selected

Power level configuration example:
http://www.libelium.com/development/waspmote/examples/dm-13-setread-power-level

6.2. Received Signal Strength Indicator

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

Example of use:

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

Related variables:

`xbeeDM.valueRSSI`→ stores the RSSI of the last received packet

The returned command value is measured in -dBm. For example if `xbeeDM.valueRSSI`=0x60, then the RSSI of the last packet received was -96dBm. 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/dm-05-get-rssi

7. 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 DigiMesh, every node in a network must have the same:*

- *PAN ID*
- *Channel*
- *Encryption configuration*

7.1. Topologies

DigiMesh provides different topologies to create a network:

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

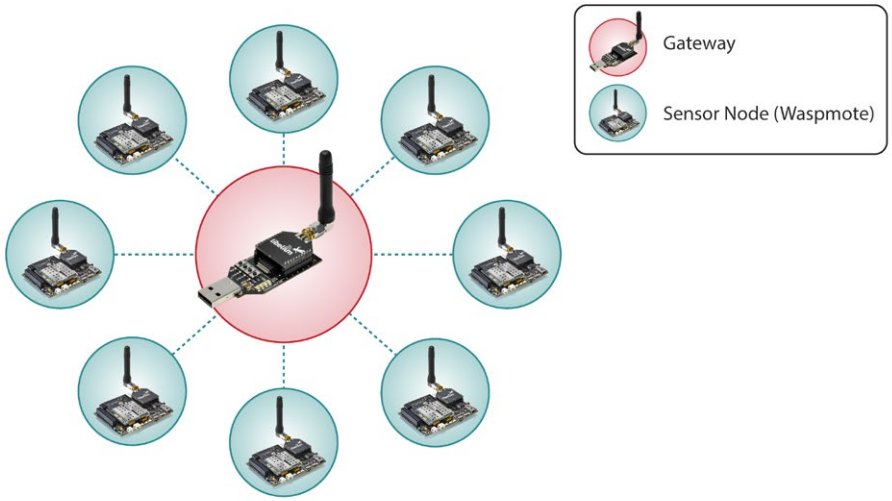


Figure : Star Topology

- **Mesh**: all the nodes are connected among them. DigiMesh provides a Network Layer to route packets among nodes out of sight.

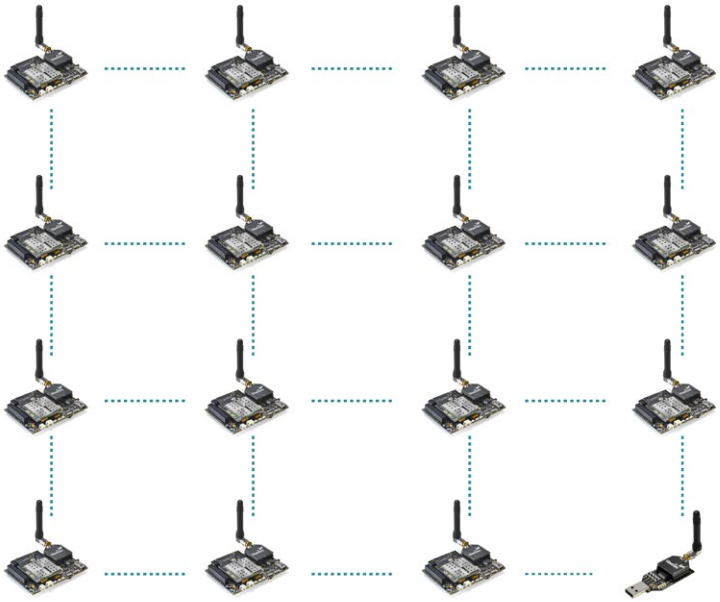


Figure : Mesh Topology

7.2. DigiMesh architecture

Mesh networking allows messages routing through several different nodes to a final destination. In the event that one RF connection between nodes is lost (due to power-loss, environmental obstructions, etc.) critical data can still reach its destination due to the mesh networking capabilities embedded inside the modules.

DigiMesh contains the following features:

- **Self-healing**: any node may enter or leave the network at any time without causing the network as a whole to fail.
- **No hierarchy and no parent-child relationships are needed.**
- **Quiet Protocol**: routing overhead will be reduced by using a reactive protocol similar to AODV.
- **Route Discovery**: rather than maintaining a network map, routes will be discovered and created only when needed.
- **Selective acknowledgments**: only the destination node will reply to route requests.
- **Reliable delivery**: reliable delivery of data is accomplished by means of acknowledgements.
- **Sleep Modes**: low power sleep modes with synchronized wake up are supported, with variable sleep and wake up times.

7.2.1. Routing

A module within a mesh network is able to determine reliable routes using a routing algorithm and table. The routing algorithm uses a reactive method derived from AODV (Ad-hoc On-demand Distance Vector). An associative routing table is used to map a destination node address with its next hop. By sending a message to the next hop address, either the message will reach its destination or be forwarded to an intermediate node which will route the message on to its destination. A message with a Broadcast address is broadcast to all neighbors. All receiving neighbors will rebroadcast the message and eventually the message will reach all corners of the network. Packet tracking prevents a node from resending a broadcast message twice.

7.2.2. Route discovery process

If the source node does not have a route to the requested destination, the packet is queued to await a route discovery (RD) process. This process is also used when a route fails. A route fails when the source node uses up its network retries without ever receiving an ACK. This results in the source node initiating RD.

RD begins by the source node broadcasting a route request (RREQ). Any node that receives the RREQ that is not the ultimate destination is called an intermediate node. Intermediate nodes may either drop or forward an RREQ, depending on whether the new RREQ has a better route back to the source node. If so, information from the RREQ is saved and the RREQ is updated and broadcast. When the ultimate destination receives the RREQ, it unicasts a route reply (RREP) back to the source node along the path of the RREQ. This is done regardless of route quality and regardless of how many times an RREQ has been seen before.

This allows the source node to receive multiple route replies. The source node selects the route with the best round trip route quality, which it will use for the queued packet and for subsequent packets with the same destination address.

7.3. Addressing

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

DigiMesh supports Unicast and Broadcast transmissions:

- **Unicast:** When transmitting in unicast communications, reliable delivery of data is accomplished using retries and acknowledgements. The number of retries is determined by the Network Retries (NR) parameter, explained in "Node parameters" chapter. RF data packets are sent up to NR + 1 times and ACKs (acknowledgements) are transmitted by the receiving node upon receipt. If a network ACK is not received within the time it would take for a packet to traverse the network twice, a retransmission occurs.
- **Broadcast:** Broadcast transmissions will be received and repeated by all routers in the network. Because ACKs are not used, the originating node will send the broadcast multiple times. By default a broadcast transmission is sent four times. Essentially the extra transmissions become automatic retries without acknowledgments. This will result in all nodes repeating the transmission four times as well. In order to avoid RF packet collisions, a random delay is inserted before each router relays the broadcast message. This time is specified by Network Delay Slots (NN), explained in "Node Parameters" chapter. Sending frequent broadcast transmissions can quickly reduce the available network bandwidth and as such should be used sparingly.

7.4. Maximum payloads

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

	Unicast	Broadcast
Encrypted	73 Bytes	73 Bytes
Un-encrypted	73 Bytes	73 Bytes

Figure : Maximum payloads size

7.5. Sending data

7.5.1. Using Waspmote Frame

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

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

7.5.2. 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";
   xbeeDM.send( rx_address, data);
}
```

- Send Waspmote Frames:

```
{
   xbeeDM.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};
   xbeeDM.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:

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

Related variables:

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

7.5.3. Examples

Send packets in unicast mode:
http://www.libelium.com/development/waspmote/examples/dm-02-send-packets

Send packets in broadcast mode:
http://www.libelium.com/development/waspmote/examples/dm-04a-send-broadcast

Send packets using the expansion board:
http://www.libelium.com/development/waspmote/examples/dm-06a-expansion-board-send

Complete example, send packets in unicast mode and wait for a response:
http://www.libelium.com/development/waspmote/examples/dm-08a-complete-example-send

7.6. Receiving data

7.6.1. 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 = xbeeDM.receivePacketTimeout( 10000 );
}
```

Related variables:

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

7.6.2. Examples

Receiving packets example:
http://www.libelium.com/development/waspmote/examples/dm-03-receive-packets

Receive packets in broadcast mode (the same procedure as if it was unicast mode):
http://www.libelium.com/development/waspmote/examples/dm-04b-receive-broadcast

Receive packets using the expansion board:
http://www.libelium.com/development/waspmote/examples/dm-06b-expansion-board-reception

Complete example, receive packets and send a response back to the sender:
http://www.libelium.com/development/waspmote/examples/dm-08b-complete-example-receive

8. Node Discovery

XBee modules provide some features for discovering and searching nodes.

8.1. 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, a 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`: Not returned in DigiMesh.
- `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 DigiMesh.
- `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 DigiMesh.

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:

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

Related variables:

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

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

8.2. 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];
   xbeeDM.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/dm-11a-node-search-tx
http://www.libelium.com/development/waspmote/examples/dm-11b-node-search-rx

8.3. 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:

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

Related variables:

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

8.4. Node Discovery Time

It is the amount of time (NT) a node will wait for responses from other nodes when performing a ND. Range: 0X20 to 0X2EE0 [x100 ms]. Default value: 0x0082.

Example of use:

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

Available information:

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

9. Sleep options

9.1. Sleep modes

All the nodes in a DigiMesh network can sleep, entering in a low power consumption mode.

There are two types of sleep modes: asynchronous or synchronous.

9.1.1. Asynchronous Cyclic Sleep

Asynchronous sleep modes can be used to control the sleep state on a module by module basis. Modules operating in an asynchronous sleep mode should not be used to route data.

9.1.2. Synchronous Cyclic Sleep

A node in synchronous cyclic sleep mode sleeps for a programmed time, wakes up in unison with other nodes, exchanges data and synchronization messages, and then returns to sleep. All synchronized cyclic sleep nodes enter and exit a low power state at the same time. This forms a cyclic sleeping network. While asleep, it cannot receive RF messages, neither will it read commands from the UART port.

9.2. Sleep parameters

There are some parameters involved in setting a node to sleep.

9.2.1. Sleep Mode

Sets the sleep mode for a module. Waspmote supports the following modes:

- 0: Normal Mode. The node will not sleep. Normal mode modules are not compatible with nodes configured for sleep. Excepting the case of adding a new node to a sleep-compatible network, a network should consist of either only normal nodes or only sleep-compatible nodes.
- 1: Asynchronous Pin Sleep Mode: Pin sleep allows the module to sleep and wake according to the state of the Sleep_RQ pin (pin 9). When Sleep_RQ is set 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 set low.
- 7: Synchronous Sleep Support Mode. The node will synchronize itself with a sleeping network but will not sleep. They are especially useful when used as preferred sleep coordinator nodes and as aids in adding new nodes to a sleeping network.
- 8: Synchronous Cyclic Sleep Mode. The node sleeps for a programmed time, wakes in unison with other nodes, exchanges data and sync messages, and then returns to sleep.

Example of use:

```
{
  xbeeDM.setSleepMode(8); //  Set Sync Cyclic Sleep Mode
  xbeeDM.getSleepMode(); // Get the Sleep Mode used
}
```

Related variables:

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

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

Cyclic sleep mode example:
http://www.libelium.com/development/waspmote/examples/dm-09-set-cyclic-sleep-mode

9.2.2. Sleep period

When setting the XBee module to a synchronous cyclic sleep mode, this parameter determines how long a node will sleep per period, with a maximum 4 hours (SP parameter). In the parent, it determines how long it will buffer a message for the sleeping device. Time in units of 10 ms represented in hexadecimal. Range: 1 - 1440000 (x 10 ms).

SP values example:

2 seconds: 0x0000C8 (default value)
5 seconds: 0x0001F4
10 seconds: 0x0003E8
15 seconds: 0x0005DC
4 hours: 0x15F900

Example of use:

```
{
  uint8_t asleep[3]={0x15,0xF9,0x00};
  xbeeDM.setSleepTime(asleep); //  Set Sleep period to 4 hours
}
```

Related variables:

`xbeeDM.sleepTime` → stores the sleep period the module will be sleeping

9.2.3. Time before sleep

When setting the XBee module to a synchronous cyclic sleep mode, this parameter determines the time a module will be awake waiting before sleeping (ST parameter). It resets each time data is received via RF or serial. Once the timer expires, the device will enter low-power state. Time in units of ms represented in hexadecimal. Range: 0x45-0x36EE80.

ST values example:

2 seconds: 0x0007D0 (default value)
5 seconds: 0x001388
10 seconds: 0x002710
15 seconds: 0x003A98
1 hour: 0x36EE80

Example of use:

```
{
  uint8_t awake[3]={0x36,0xEE,0x80};
  xbeeDM.setAwakeTime(awake); //  Set time the module remains awake: 1hour
}
```

Related variables:

`xbeeDM.awakeTime` → stores the time the module remains awake

9.2.4. Sleep options

It configures options for sleeping modes. Possible values are read as a bitmask (Bit 0 and Bit 1 cannot be set at the same time).

For synchronous sleep modules, the following sleep options are defined:

- bit 0: Preferred sleep coordinator. The node always acts as sleep coordinator
- bit 1: Non-sleep coordinator. Node never acts as a sleep coordinator
- bit 2: Enable API sleep status messages
- bit 3: Disable early wake-up for missed synchronizations
- bit 4: Enable node type equality
- bit 5: Disable coordinator rapid synchronization deployment mode. For asynchronous sleep modules, the following sleep options are defined:
- bit 8: Always wake for ST time

Example of use:

```
{
  xbeeDM.setSleepOptions(0x01); //  Set the node to be the preferred sleep coordinator
}
```

Related variables:

`xbeeDM.sleepOptions` → stores the sleep option chosen

10. Synchronizing the network

Sleeping routers under DigiMesh allow to all nodes in the network to synchronize their sleep and wake up times. All synchronized nodes enter and exit into low power state at the same time. This forms a cyclic sleeping network. Nodes synchronize by receiving a special RF packet called 'synch' message which is sent by a sleep coordinator. Any node in the network can become an sleep coordinator through a process called nomination. The sleep coordinator will send one synch message at the beginning of each wake up period. The synch message to broadcast category is repeated by each node in the network, since it belongs.

10.1. Operation

One node in a sleeping network acts as the sleeping coordinator. At the beginning of a wake cycle the sleep coordinator will send a sync message as a broadcast to all nodes in the network. This message contains synchronization information and the wake and sleep times for the current cycle. All cyclic sleep nodes receiving a sync message will remain awake for the wake time and then sleep for the sleep period specified.

10.1.1. Synchronization messages

Nodes which have not been synchronized or that have lost sync will send messages requesting sync information. Synchronized nodes which receive one of these messages will respond with a synchronization packet.

Deployment mode (set by default) is used by sleep compatible nodes when they are first powered up and the sync message has not been relayed. A sleep coordinator in deployment mode will rapidly send sync messages until it receives a relay of one of those messages. If a node which has exited deployment mode receives a sync message from a sleep coordinator which is in deployment mode, the sync will be rejected and a corrective sync will be sent to the sleep coordinator. Deployment mode can be disabled using the sleep options command (SO).

A sleep coordinator which is not in deployment mode or which has had deployment mode disabled will send a sync message at the beginning of the wake cycle The sleep coordinator will then listen for a neighboring node to relay the sync. If the relay is not heard, the sync coordinator will send the sync one additional time.

10.2. Becoming a sleep coordinator

A node can become a sleep coordinator in one of several ways:

10.2.1. Preferred sleep coordinator

A node can be specified to always act as a sleep coordinator. This is done by setting the preferred sleep coordinator bit (SO=1). Besides, it is necessary to set the synchronous sleep support mode (SM=7) in order to synchronize with the rest of the network nodes.

A node with the sleep coordinator bit set will always send a sync message at the beginning of a wake cycle.

For this reason, it is imperative that no more than one node in the network has this bit set. Although it is not necessary to specify a preferred sleep coordinator, it is often useful to select a node for this purpose to improve network performance.

The preferred sleep coordinator bit should be used with caution. The advantages of using the option become weaknesses when used on a node that is not positioned or configured properly.

10.2.2. Nomination and election

Nomination is the process where a node becomes a sleep coordinator at the start of a sleeping network or in the event a sleep coordinator fails as a replacement. This process is automatic with any sleeping node being eligible to become the sleep coordinator for the network. This process can be managed through 'Sleep Options' by allowing a node to alter the algorithm used for nomination giving the node a greater chance to become the sleep coordinator. This option is designed for maintenance purposes and is not necessary for cyclic sleep operation.

If the node has the preferred sleep coordinator option-enabled ('Sleep Options'=1), then it will poll during its first cycle for a synch message. If it becomes synched by receiving a synch message, it will cycle back to sleep and will not become the sleep coordinator. If it does not become synched during that first cycle, then it will nominate itself as the sleep coordinator and will start sending synch messages at the start of its second cycle.

Any sleeping node, if it does not receive a message for three cycles, may nominate itself to act as a replacement sleep coordinator.

Depending on the platform and other configured options, such a node will eventually nominate itself after a number of cycles without a sync. A nominated node will begin acting as the new network sleep coordinator. If multiple nodes nominate themselves at the same time, then an election will take place to resolve which node will function as network's sleep coordinator. A node will disable synching if it receives a synch message from a senior node. A node running in normal mode (with preferred sleep coordinator option enabled) is senior to any node operating in sleep mode. A node with the largest MAC address value is senior to any other node operating in the same mode.

10.2.3. Changing sleep parameters

Any sleep compatible node in the network which does not have the non-sleep coordinator sleep option set can be used to make changes to the network's sleep and wake times. If a node's SP and/or ST are changed to values different from those that the network is using, that node will become the sleep coordinator. That node will begin sending sync messages with the new sleep parameters at the beginning of the next wake cycle.

10.3. Configuration

10.3.1. Starting a sleeping network

By default, all new nodes operate in normal (non-sleep) mode. To start a sleeping network, follow these steps:

1. Enable the preferred sleep coordinator option on one of the nodes (SO=1), and set its the sleep compatible mode (SM=7). Sleep/Awake periods (SP and ST parameters) should be set to a quick cycle time.
2. Next, power on the new nodes within range of the sleep coordinator. The nodes will quickly receive a sync message and synchronize themselves to the short cycle SP and ST.
3. Configure the new nodes in their desired sleep mode as cyclic sleeping nodes (SM=8) or sleep support nodes (SM=7).
4. Set the SP and ST values on the sleep coordinator to the desired values for the deployed network.
5. Wait a cycle for the sleeping nodes to sync themselves to the new SP and ST values.
6. Disable the preferred sleep coordinator option bit on the sleep coordinator (unless a preferred sleep coordinator is desired).
7. Deploy the nodes to their positions.

10.3.2. Adding a new node to an existing network

To add a new node to the network, the node must receive a sync message from a node already in the network. On power-up, an unsynchronized sleep compatible node will periodically send a broadcast requesting a sync message and then sleep for its SP period. Any node in the network that receives this message will respond with a sync. Because the network can be asleep for extended periods of time, and as such cannot respond to requests for sync messages, there are methods that can be used to sync a new node while the network is asleep. e.g. Power the new node on within range of a sleep support node. Sleep support nodes are always awake and will be able to respond to sync requests promptly.

10.3.3. Changing sleep parameters

Changes to the sleep and wake cycle of the network can be made by changing the SP and/or ST of the sleep coordinator. If the sleep coordinator is not known, any node that does not have the non-sleep coordinator sleep option bit set, can be used. When changes are made to a node's sleep parameters, that node will become the network's sleep coordinator (unless it has the non-sleep coordinator option selected) and will send a sync message with the new sleep settings to the entire network at the beginning of the next wake cycle. The network will immediately begin using the new sleep parameters after this sync is sent.

Changing sleep parameters increases the chances that nodes will lose sync. If a node does not receive the sync message with the new sleep settings, it will continue to operate on its old settings. To minimize the risk of a node losing sync and to facilitate the resyncing of a node that does lose sync, the following precautions can be taken:

1. Whenever possible, avoid changing sleep parameters.
2. Enable the missed sync early wake up sleep option (SO). This command is used to tell a node to wake up progressively earlier based on the number of cycles it has gone without receiving a sync. This will increase the probability that the unsynced node will be awake when the network wakes up and sends the sync message. ***Note:*** *using this sleep option increases reliability but may decrease battery life.*

3. When changing between two sets of sleep settings, choose settings so that the wake periods of the two sleep settings will happen at the same time. In other words, try to satisfy the following equation: (SP1 + ST1) = N * (SP2 + ST2), where SP1/ST1 and SP2/ST2 are the desired sleep settings and N is an integer.

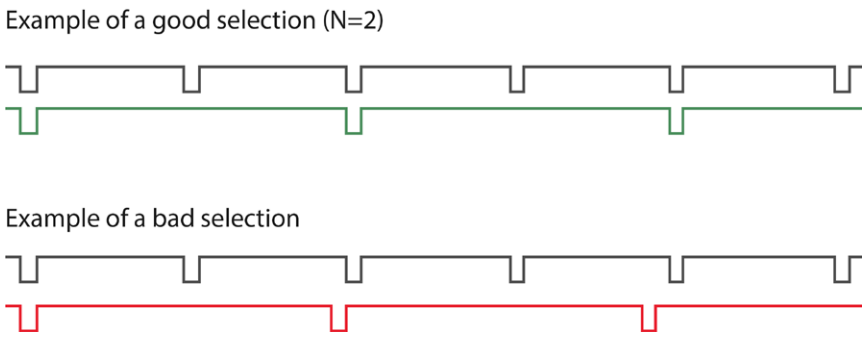


Figure : Sleep setting examples

10.3.4. Rejoining nodes which have lost sync

It is recommended to build the network with redundant mesh nodes to increase robustness. If a scenario exists such that the only route connecting a subnet to the rest of the network depends on a single node and that node fails, then multiple subnets may arise while using the same wake/sleep intervals. When this occurs the first task is to repair, replace, and strengthen the weak link with new and/or redundant modules to fix the problem and prevent it from occurring in the future. Subnets can drift out of phase with each other if the network is configured in one of the following ways:

- If multiple modules in the network have had the non-sleep coordinator sleep option bit disabled and are thus eligible to be nominated as a sleep coordinator.
- If the modules in the network are not using the auto early wake-up sleep option.

If a network has multiple subnets that have drifted out of phase with each other, get the subnets back in phase with the following steps:

1. Place a sleep support node in range of both subnets.
2. Select a node in the subnet that you want the other subnet to sync up with. Use this node to slightly change the sleep cycle settings of the network (increment ST, for example).
3. Wait for the subnet's next wake cycle. During this cycle, the node selected to change the sleep cycle parameters will send the new settings to the entire subnet it is in range of, including the sleep support node which is in range of the other subnet.
4. Wait for the out-of-sync subnet to wake up and send a sync. When the sleep support node receives this sync, it will reject it and send a sync to the subnet with the new sleep settings.
5. The subnets will now be in sync. The sleep support node can be removed. If desired, the sleep cycle settings can be changed back to what they were.

In the case that only a few nodes need to be replaced, this method can also be used:

1. Reset the out-of-sync node and set its sleep mode to cyclic sleep. Set it up to have a short sleep cycle.
2. Place the node in range of a sleep support node or wake a sleeping node.
3. The out-of-sync node will receive a sync from the node which is synchronized to the network and sync to the network sleep settings.

11. Security and data encryption

11.1. DigiMesh security and Data encryption overview

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

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

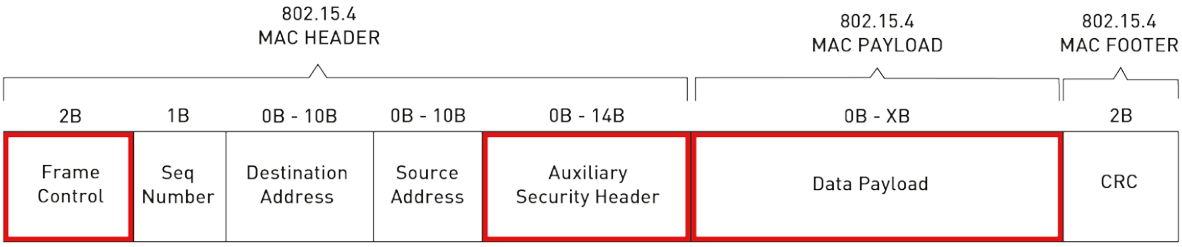


Figure : IEEE 802.15.4 frame

11.2. Security in API libraries

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

11.2.1. Encryption enable

Enables the 128-bit AES encryption in the modules.

Example of use:

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

Related variables:

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

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

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

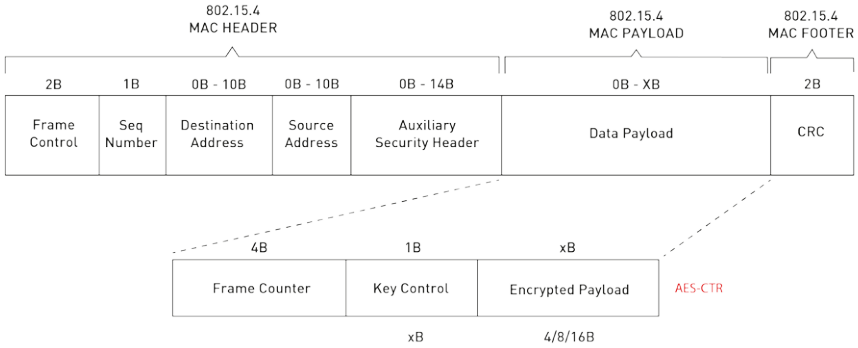


Figure : AES-CTR encryption frame

11.2.2. 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!"
  xbeeDM.setLinkKey(KEY); // Set Encryption Key
}
```

Related variables:

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

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

11.3. 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 will not be possible to communicate between different XBee modules.

12. Code examples and extended information

In the Waspmote Development section you can find complete examples:

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

Example:

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

// Destination MAC address
////////////////////////////////////////
char RX_ADDRESS[] = “0013A200402C02E3”;
////////////////////////////////////////

// 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
  xbeeDM.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 = xbeeDM.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);
}
```

13. API changelog

Keep track of the software changes on this link:

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