INDEX

1. General Considerations ................................................................. 5
   1.1. Wasp mote Libraries ................................................................. 5
   1.1.1. Wasp mote XBee Files ............................................................ 5
   1.1.2. Constructor ........................................................................... 5
   1.2. API Functions .......................................................................... 5
   1.3. API extension .......................................................................... 6
   1.4. Wasp mote reboot .................................................................... 6
   1.5. Constants pre-defined .............................................................. 6

2. Initialization .................................................................................. 7
   2.1. Initializing ................................................................................ 7
   2.2. Setting ON .............................................................................. 7
   2.3. Setting OFF ............................................................................ 8

3. Node Parameters .......................................................................... 8
   3.1. MAC Address .......................................................................... 8
   3.2. Network Address ................................................................... 8
   3.3. PAN ID .................................................................................. 9
   3.4. Node Identifier ....................................................................... 9
   3.5. Channel ................................................................................ 9

4. Packet Parameters ....................................................................... 11
   4.1. Structure used in packets ......................................................... 11
   4.2. Maximum payloads ................................................................. 13
   4.3. 802.15.4 Compliance ............................................................... 14

5. Power Gain and Sensibility .......................................................... 15
   5.1. Power Level .......................................................................... 15
   5.2. Received Signal Strength Indicator ......................................... 16

6. Radio Channels ........................................................................... 17
   6.1. Scan Channels ....................................................................... 17
   6.2. Energy Scan .......................................................................... 17
   6.3. Random Delay ....................................................................... 18
   6.4. CCA Threshold ...................................................................... 18
   6.5. CCA Failures ......................................................................... 18
   6.6. ACK Failures ......................................................................... 19
   6.7. Retries .................................................................................. 19

7. Connectivity ................................................................................ 20
   7.1. Topologies ............................................................................ 20
   7.2. Connections ......................................................................... 20
   7.2.1. Unicast .............................................................................. 20
# Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.2. Broadcast</td>
<td>20</td>
</tr>
<tr>
<td>7.3. Sending Data</td>
<td>20</td>
</tr>
<tr>
<td>7.3.1. API Frame Structure</td>
<td>20</td>
</tr>
<tr>
<td>7.3.2. Application Header</td>
<td>21</td>
</tr>
<tr>
<td>7.3.3. Fragmentation</td>
<td>22</td>
</tr>
<tr>
<td>7.3.4. Structure packetXBee</td>
<td>22</td>
</tr>
<tr>
<td>7.3.5. Sending without API header</td>
<td>22</td>
</tr>
<tr>
<td>7.3.6. Examples</td>
<td>22</td>
</tr>
<tr>
<td>7.4. Receiving Data</td>
<td>25</td>
</tr>
<tr>
<td>7.4.1. Examples</td>
<td>26</td>
</tr>
<tr>
<td>7.5. Discovering and Searching Nodes</td>
<td>28</td>
</tr>
<tr>
<td>7.5.1. Structure used in Discovery</td>
<td>28</td>
</tr>
<tr>
<td>7.5.2. Searching specific nodes</td>
<td>29</td>
</tr>
<tr>
<td>7.5.3. Node discovery to a specific node</td>
<td>29</td>
</tr>
<tr>
<td>8. Starting a Network</td>
<td>30</td>
</tr>
<tr>
<td>8.1. Choosing a channel</td>
<td>30</td>
</tr>
<tr>
<td>8.1.1. Random Channel</td>
<td>30</td>
</tr>
<tr>
<td>8.1.2. Channel based on Energy Channel</td>
<td>30</td>
</tr>
<tr>
<td>8.2. Choosing a PAN ID</td>
<td>30</td>
</tr>
<tr>
<td>9. Joining an Existing Network</td>
<td>32</td>
</tr>
<tr>
<td>9.1. Network parameters are known</td>
<td>32</td>
</tr>
<tr>
<td>9.1.1. Channel</td>
<td>32</td>
</tr>
<tr>
<td>9.1.2. PAN ID</td>
<td>32</td>
</tr>
<tr>
<td>9.2. Network parameters are unknown</td>
<td>32</td>
</tr>
<tr>
<td>9.2.1. PAN ID unknown &amp; Channel known</td>
<td>32</td>
</tr>
<tr>
<td>9.2.2. Channel unknown &amp; PAN ID known</td>
<td>33</td>
</tr>
<tr>
<td>9.2.3. Channel &amp; PAN ID unknown</td>
<td>33</td>
</tr>
<tr>
<td>9.3. Node Discovery</td>
<td>34</td>
</tr>
<tr>
<td>9.3.1. Node Discovery</td>
<td>34</td>
</tr>
<tr>
<td>9.3.2. Node Discovery Time</td>
<td>34</td>
</tr>
<tr>
<td>9.3.3. Node Discover Options</td>
<td>35</td>
</tr>
<tr>
<td>9.3.4. Questions related with Discovering Nodes</td>
<td>35</td>
</tr>
<tr>
<td>10. Sleep Options</td>
<td>37</td>
</tr>
<tr>
<td>10.1. Sleep Mode</td>
<td>37</td>
</tr>
<tr>
<td>10.2. Pin Hibernate Mode</td>
<td>37</td>
</tr>
<tr>
<td>10.3. Pin Doze Mode</td>
<td>37</td>
</tr>
<tr>
<td>10.4. ON/OFF Modes</td>
<td>38</td>
</tr>
<tr>
<td>11. Security and Data Encryption</td>
<td>38</td>
</tr>
<tr>
<td>11.1. IEEE 802.15.4 Security and Data encryption Overview</td>
<td>38</td>
</tr>
<tr>
<td>11.2. Security in API libraries</td>
<td>38</td>
</tr>
<tr>
<td>11.2.1. Encryption Enable</td>
<td>38</td>
</tr>
<tr>
<td>11.2.2. Encryption Key</td>
<td>39</td>
</tr>
<tr>
<td>11.2.3. Maximum Data Payloads</td>
<td>39</td>
</tr>
</tbody>
</table>
11.3. Security in a network .............................................................................................................................. 40
    11.3.1. Creating a network enabling security .............................................................................................. 40
    11.3.2. Joining a security enabled network ............................................................................................... 40

12. Code examples and extended information ................................................................................................. 41
1. General Considerations

1.1. Waspmote Libraries

1.1.1. Waspmote XBee Files

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

1.1.2. Constructor

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

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

1.2. 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

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

Related Variables

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

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

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

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

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

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

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

Example of use

```c
{
  xbee802.writeValues(); // Keep values after power down
}
```
1.3. 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

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

Related Variables

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

1.4. 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.

1.5. 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 100 bytes, 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.
2. 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.

2.1. 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.
- **encryptMode**: specifies if encryption mode is enabled.
- **powerLevel**: specifies the power transmission level.
- **timeRSSI**: specifies the time RSSI LEDs are on.
- **sleepOptions**: specifies the options for sleeping.
- **retries**: specifies the number of retries to execute in addition to the three retries defined in the 802.15.4 protocol.
- **delaySlots**: specifies the minimum value of the back-off exponent in CSMA/CA.
- **macMode**: specifies the Mac Mode used.
- **energyThreshold**: specifies the energy threshold used to determine if the channel is free.
- **counterCCA**: specifies the number of times too much energy was found in the channel.
- **counterACK**: specifies the number of times an ACK was lost.

Example of use:

```c
{xbee802.init(XBEE_802_15_4,FREQ2_4G,NORMAL); // initializes the variables}
```

2.2. 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:

```c
{xbee802.ON(); // Opens the UART and switches the XBee ON}
```
2.3. Setting OFF

It closes the UART and switches the XBee OFF.

Example of use

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

3. 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.

3.1. MAC Address

A 64-bit RF module’s unique IEEE address. It is divided in two groups of 32 bits (High and Low). It identifies uniquely a node inside a network due to it can not be modified and it is given by the manufacturer. It is used in 64-bit unicast transmissions.

Example of use

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

Related Variables

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

3.2. Network Address

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

Example of use

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

Related Variables

- `xbee802.sourceNA[0-1]` → stores the 16-bit network address
3.3. PAN ID

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

Example of use

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

Related Variables

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

3.4. Node Identifier

A max 20-character ASCII string 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

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

Related Variables

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

3.5. Channel

This parameter defines the frequency channel used by the module to transmit and receive. 802.15.4 defines 16 channels to be used.

- 2.40-2.48GHz : 16 channels

This module works in 2.4GHz band, having 16 channels with a 5MHz bandwidth per channel.

2.4GHz Band

![Operating Frequency Bands](image)

Figure 3.1: Operating Frequency Bands
### Node Parameters

#### Example of use

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

#### Related Variables

- `xbee802.channel` → stores the operating channel

---

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

*Figure 3.2: Channel Frequency Numbers*
4. Packet Parameters

4.1. 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:

```c
// ********** 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**
  
  A 64-bit destination address. It is used to specify the MAC address of the destination node. It is used in 64-bit unicast transmissions.

- **mode**
  
  Transmission mode chosen to transmit that packet. Available values:
  
  - 0 : Unicast transmission
  - 1 : Broadcast transmission
  - 2 : Application binding transmission, using clusters and endpoints (Not used in this protocol)
  - 3 : Synchronization packet
Packet Parameters

- **address_type**
  Address type chosen, between 16-bit and 64-bit addresses.
  - 0 : 16-bit address
  - 1 : 64-bit address

- **nAD**
  A 16-bit Destination Network Address. It is used in 16-bit unicast 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**
  Not used in 802.15.4.

- **DE**
  Not used in 802.15.4.

- **CID**
  Not used in 802.15.4.

- **PID**
  Not used in 802.15.4.

- **MY_known**
  Not used in 802.15.4.

- **opt**
  Options for the transmitted packet. Options available in 802.15.4:
  - 0x01: Disable ACK.
  - 0x04 : Send packet with Broadcast PAN ID

- **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**
  A 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**
  A 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**
  A 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**
  A 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.
Packet Parameters

• **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 what 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 media of all received fragments RSSI.

• **address_typeS**
  Address type used to send the packet by the transmitter.
  • 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**
  Relative position of the fragment regarding the entire message.

• **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**
  Not used in 802.15.4.

• **discov_status**
  Not used in 802.15.4.

• **true_naD**
  Not used in 802.15.4.

• **retries**
  Not used in 802.15.4.

4.2. Maximum payloads

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

<table>
<thead>
<tr>
<th></th>
<th>@16bit Unicast</th>
<th>@64bit Unicast</th>
<th>Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encrypted</strong></td>
<td>98Bytes</td>
<td>94Bytes</td>
<td>95Bytes</td>
</tr>
<tr>
<td><strong>Un-Encrypted</strong></td>
<td>100Bytes</td>
<td>100Bytes</td>
<td>100Bytes</td>
</tr>
</tbody>
</table>

*Figure 4.1: Maximum Payload Size*
4.3. 802.15.4 Compliance

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

These features are:

• Node Discover: explained in chapter 6. It performs a scan in the network and reports other nodes working on the network.
• Destination Node: explained in chapter 6. It performs a scan in the network to look for a specific node.
• Duplicate Detection: 802.15.4 does not detect duplicate packets. Using Digi extra header, this problem is solved, discarding automatically the duplicate packets. If Digi header is not enabled, the application level will have to deal with this problem.

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

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

Example of use:

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

Related Variables

`xbee802.macMode` → stores the Mac Mode selected
5. Power Gain and Sensibility

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

5.1. Power Level

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>XBee</th>
<th>XBee-PRO</th>
<th>XBee-PRO International</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-10dBm</td>
<td>10dBm</td>
<td>PL=4 : 10dBm</td>
</tr>
<tr>
<td>1</td>
<td>-6dBm</td>
<td>12dBm</td>
<td>PL=3 : 8dBm</td>
</tr>
<tr>
<td>2</td>
<td>-4dBm</td>
<td>14dBm</td>
<td>PL=2 : 2dBm</td>
</tr>
<tr>
<td>3</td>
<td>-2dBm</td>
<td>16dBm</td>
<td>PL=1 : -3dBm</td>
</tr>
<tr>
<td>4</td>
<td>0dBm</td>
<td>18dBm</td>
<td>PL=0 : -3dBm</td>
</tr>
</tbody>
</table>

Figure 5.1: 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 5.2: XBee Output Power Level](image1.png)

![Figure 5.3: XBee-PRO Output Power Level](image2.png)

Example of use

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

Related Variables

xbee802.powerLevel → stores the power output level selected
5.2. Received Signal Strength Indicator

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

Example of use:

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

Related Variables

- `xbee802.valueRSSI` → stores the RSSI of the last received packet

The ideal working mode is getting the maximum coverage with the minimum power level. Thereby, a compromise between power level and coverage appears. Each application scenario will need some tests to find the best combination of both parameters.
6. Radio Channels

2.4GHz Band

![Frequency Channels on 2.4GHz Band](image)

The XBee 802.15.4 module works on 2.4GHz band, so there are 16 channels available. The values are between 0x0B-0x1A(16 values).

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

6.1. Scan Channels

List of channels to scan for all Active and Energy Scans as a bitfield. Setting this parameter to 0xFFFF, will cause all the channels to be scanned.

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

Example of use

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

Related Variables

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

6.2. Energy Scan

The maximum energy on each channel is returned, starting with the first channel selected in Scan Channels parameter. The amount of time this energy scan is performed is specified as an API function input. This time is calculated in this way $2^{ED} * 15.36$ms.

The recommended process for choosing a free channel is:

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

Example of use:

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

Related Variables

`xbee802.energyChannel[0-15]` → stores the energy found on each scanned channel
NOTE: The amount of time specified to scan each channel may affect the detected energy. In our tests, selecting the minimum value was not sufficient to detect the energy, so it is recommended to select a higher value.

NOTE2: After testing, a usual energy value for a free channel is around **-84dBm** and **-37dBm** for an occupied one. When the energy scan is performed, these values may be used to determine whether a channel is occupied or not.

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

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

### 6.3. Random Delay

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

Example of use:

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

**Related Variables**

- `xbee802.delaySlots` → stores the exponent

### 6.4. CCA Threshold

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

Example of use:

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

**Related Variables**

- `xbee802.energyThreshold` → stores the energy threshold

### 6.5. CCA Failures

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

Example of use:

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

Related Variables

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

### 6.6. ACK Failures

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

Example of use:

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

Related Variables

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

### 6.7. Retries

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

Example of use:

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

Related Variables

```
xbee802.retries[0-1] → stores the retries
```

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

7.1. Topologies

802.15.4 provides a p2p topology to create a network:

- **Peer-to-Peer (p2p):** networks can form arbitrary patterns of connections, and their extension is only limited by the distance between each pair of nodes.

![Peer-to-Peer Topology](image)

7.2. Connections

Every RF data packet sent over-the-air contains a Source Address and Destination Address field in its header. The RF module conforms to the 802.15.4 specification and supports both short 16-bit addresses and long 64-bit addresses. A unique 64-bit IEEE source address is assigned at the factory and can be read with the functions explained in chapter 2. Short addressing must be configured manually.

IEEE 802.15.4 supports unicast and broadcast transmission.

7.2.1. Unicast

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

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

7.2.2. Broadcast

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

7.3. Sending Data

Sending data is a complex process which needs some special structures and functions to carry out. Due to the limit on maximum payloads (see chapter 3) it is usual the packet has to be fragmentation. An application header has been created to deal with this process. This application header is sent inside RF Data, following the API Frame Structure.

7.3.1. API Frame Structure

The structure used to transmit packets via RF is specified by the used modules. Depends on using 64-bit or 16-bit transmissions, API Frame Structure will be different. The application header has been added to that defined structured as shown below:
7.3.2. Application Header

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

To add some application level features to the receiver, some fields have been added like 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 answer. To identify this node, three ID 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 data field used in an API frame is represented in figure 6.4.

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

7.3.3. 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 without even transmitting the packet.

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

7.3.4. Structure packetXBee

It is the structure used for the packet to send explained in chapter 3.

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 the process success or failure.

7.3.5. 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.

7.3.6. Examples

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

• Sending a packet of 50 bytes without API header

```c
{
    char* data;
    for(int c=0;c<50;c++) // Set the data
    {
        data[c]='A';
    }
    xbee802.send("0013A2004030F66A",data);
}
```
• Sending a packet of 50 bytes. Unicast 16-bit Mode. 16-bit NA Source ID.

```c
{ 
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->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';
}
xbee802.setOriginParams(paq_sent, "1221", MY_TYPE); // sets Origin Parameters
xbee802.setDestinationParams(paq_sent, "1234", data, MY_TYPE, DATA_ABSOLUTE); // sets Destination Parameters
state=xbee802.sendXBee(paq_sent); // Call function responsible to send data.
free(paq_sent);
paq_sent=NULL;
}
```

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

<table>
<thead>
<tr>
<th>Application ID</th>
<th>Fragment Number</th>
<th>First Fragment Indicator [#]</th>
<th>Source Type ID</th>
<th>Source ID</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x52</td>
<td>1</td>
<td>#</td>
<td>0</td>
<td>0x12 0x21</td>
<td>A A A ... A A (50 Bytes)</td>
</tr>
</tbody>
</table>

*Figure 7.6 : API Application Header*

• Sending a 50-bytes packet. Unicast 64-bit Mode. 64-bit MAC Source ID.

```c
{ 
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->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';
}
xbee802.setOriginParams(paq_sent, "0013A2004030F66A", MAC_TYPE);
xbee802.setDestinationParams(paq_sent, "0013A2004030F686", data, MAC_TYPE, DATA_ABSOLUTE);
state=xbee802.sendXBee(paq_sent); // Call function responsible to send data
free(paq_sent);
paq_sent=NULL;
}
```

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

<table>
<thead>
<tr>
<th>Application ID</th>
<th>Fragment Number</th>
<th>First Fragment Indicator [#]</th>
<th>Source Type ID</th>
<th>Source ID</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x52</td>
<td>1</td>
<td>#</td>
<td>1</td>
<td>0x00 0x13 0xA2 0x00 0x40 0x30 0xF6 0x86</td>
<td>A A A ... A A (50 Bytes)</td>
</tr>
</tbody>
</table>

*Figure 7.7 : API Application Header*
• **Sending a 50-bytes packet. Broadcast Mode. Node Identifier Source ID Type.**

```c
{  
    packetXBee* paq_sent; // create packet to send  
    paq_sent=(packetXBee*) malloc(1,sizeof(packetXBee));  
    paq_sent->mode=BROADCAST; // set Broadcast mode  
    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';  
    }
    xbee802.setOriginParams(paq_sent, “forrestNode-01”, NI_TYPE);  
    xbee802.setDestinationParams(paq_sent, “000000000000FFFF”, data, MAC_TYPE, DATA_ABSOLUTE);  
    state=xbee802.sendXBee(paq_sent); // Call function responsible to send data  
    free(paq_sent);  
    paq_sent=NULL;  
}
```

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

![Figure 7.8: API Application Header](attachment:image.png)

• **Sending a 200-bytes packet. Unicast 64-bit Mode. 16-bit Source ID Type.**

```c
{  
    packetXBee* paq_sent; // create packet to send  
    paq_sent=(packetXBee*) malloc(1,sizeof(packetXBee));  
    paq_sent->mode=UNICAST; // set Broadcast mode  
    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';  
    }
    xbee802.setOriginParams(paq_sent, “1221”, MY_TYPE);  
    xbee802.setDestinationParams(paq_sent, “0013A2004030F686”, data, MAC_TYPE, DATA_ABSOLUTE);  
    state=xbee802.sendXBee(paq_sent); // Call function responsible to send data  
    free(paq_sent);  
    paq_sent=NULL;  
}
```

Due to the amount of data, the packet needs to be fragmented before being sent. The maximum data payload is 100Bytes (see chapter 3) and the application header will occupy a maximum of 6 Bytes (in the first fragment), so there will be three fragments, sending 94Bytes(data) in the first fragment, 95Bytes(data) in the second fragment and 11Bytes(data) in the third one.
7.4. Receiving Data

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

Before any packet has been received, 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 are 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).

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

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

```c
{    state=xbee802.treatData(); // Read a packet when XBee has noticed it to us
    if( xbee802.pos>0 )
    {
        /* Available info: (there are more available information) */
        network_address[0]=packet_finished[xbee802.pos-1]->naS[0];
        network_address[1]=packet_finished[xbee802.pos-1]->naS[1];
    }
}
```

**Available Information**

- `packet_finished[xbee802.pos-1]->naS`: stores the 16-bit network address of the sender.
- `packet_finished[xbee802.pos-1]->naO`: stores the 16-bit network address of the origin sender.
- `packet_finished[xbee802.pos-1]->mode`: stores the transmission mode. Unicast in this case.
- `packet_finished[xbee802.pos-1]->data`: stores the data received. Max size 'MAX_DATA'.
- `packet_finished[xbee802.pos-1]->packetID`: stores the packet ID of the received message. 0x52 in this case.
- `packet_finished[xbee802.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 mode. 64-bit MAC Address Source ID Type**

```c
{    state=xbee802.treatData(); // Read a packet when XBee has noticed it to us
    if( xbee802.pos>0 )
    {
        /* Available info: (there are more available information) */
        source_address[0]=packet_finished[xbee802.pos-1]->macSH[0];
        ...
        source_address[3]=packet_finished[xbee802.pos-1]->macSH[3];
    }
}
```

**Available Information**

- `packet_finished[xbee802.pos-1]->macSH`: stores the 32 upper bits of MAC sender address.
- `packet_finished[xbee802.pos-1]->macSL`: stores the 32 lower bits of MAC sender address.
- `packet_finished[xbee802.pos-1]->macOH`: stores the 32 upper bits of MAC origin sender address.
- `packet_finished[xbee802.pos-1]->macOL`: stores the 32 lower bits of MAC origin sender address.
- `packet_finished[xbee802.pos-1]->mode`: stores the transmission mode. Unicast in this case.
- `packet_finished[xbee802.pos-1]->data`: stores the data received. Max size 'MAX_DATA'.
- `packet_finished[xbee802.pos-1]->packetID`: stores the packet ID of the received message. 0x52 in this case.
- `packet_finished[xbee802.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 64-bit mode. Node Identifier Source ID Type**

```c
{    state=xbee802.treatData(); // Read a packet when XBee has noticed it to us
```
if ( xbee802.pos>0 )
{
    /* Available info: (there are more available information) */
    source_address[0]=packet_finished[xbee802.pos-1]->macSH[0];
    ...
    source_address[3]=packet_finished[xbee802.pos-1]->macSH[3];
}

Available Information

packet_finished[xbee802.pos-1]->macSH : stores the 32 upper bits of MAC sender address.
packet_finished[xbee802.pos-1]->macSL : stores the 32 lower bits of MAC sender address.
packet_finished[xbee802.pos-1]->niO : stores the 20-byte max string used to Node Identifier of the origin sender
packet_finished[xbee802.pos-1]->mode : stores the transmission mode. Unicast in this case.
packet_finished[xbee802.pos-1]->data : stores the data received. Max size ‘MAX_DATA’.
packet_finished[xbee802.pos-1]->packetID : stores the packet ID of the received message. 0x52 in this case.
packet_finished[xbee802.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 64-bit Mode. 16-bit NA Source ID Type

}  
state=xbee802.treatData(); // Read a packet when XBee has noticed it to us
if( xbee802.pos>0 )
{
    /* Available info: (there are more available information) */
    network_address[0]=packet_finished[xbee802.pos-1]->naS[0];
    network_address[1]=packet_finished[xbee802.pos-1]->naS[1];
}

Available Information

packet_finished[xbee802.pos-1]->macSH : stores the 32 upper bits of MAC sender address.
packet_finished[xbee802.pos-1]->macSL : stores the 32 lower bits of MAC sender address.
packet_finished[xbee802.pos-1]->naO : stores the 16-bit network address of the origin sender.
packet_finished[xbee802.pos-1]->mode : stores the transmission mode. Unicast in this case.
packet_finished[xbee802.pos-1]->data : stores the data received. Max size ‘MAX_DATA’.
packet_finished[xbee802.pos-1]->packetID : stores the packet ID of the received message. 0x52 in this case.
packet_finished[xbee802.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 having in a receiver node more than 1500Bytes. It is referred to a single big packet or many smaller packets in parallel.
7.5. Discovering and Searching Nodes

As explained in chapter 3, an extra Digi header enables some features for discovering and searching nodes. These features added to 802.15.4 allow a node to send a broadcast message to discover other nodes in the network within its coverage.

7.5.1. 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:

- **MY**: 16-bit Network Address of the reported module.
- **SH & SL**: 64-bit MAC Source Address of the reported module.
- **NI**: Node Identifier of the reported module.
- **PMY & DT & ST & PID & MID**: Not used in 802.15.4.
- **RSSI**: Received Signal Strength Indicator of the received packet. This parameter is quite a lot important since it indicates the link quality between the two nodes.

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

Example of use:

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

Related variables

- `xbee802.totalScannedBrothers` → stores the number of brothers have been discovered.
- `xbee802.scannedBrothers` → 'Node' structure array that stores in each position the info related to each found node. For example, `xbee802.scannedBrothers[0].MY` should store the 16-bit Network Address of the first found node.
7.5.2. 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

```c
{    packetXBee* paq_sent;
    xbee802.nodeSearch("forrestNode-01",14,paq_sent); // ‘14’ is string length
}
```

Related variables

- `paq_sent` → Stores the 16-bit and 64-bit addresses of the searched. For example

7.5.3. 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

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

Related variables

- `xbee802.totalScannedBrothers` → stores the number of brothers have been discovered. In this case its value will be ‘1’
- `xbee802.scannedBrothers` → ‘Node’ structure array that stores in each position the info related to each found node. It will store in the first position of the array the information related to the found brother
8. Starting a Network

To create a network only two parameters are necessary: channel and **PAN ID**. These parameters are the base of a network and we need to be careful choosing them. There are more parameters used to create a network like security parameters, which are not necessary but recommended (see chapter 11).

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

8.1. Choosing a channel

There are two possibilities for choosing a channel: choose a random channel or choose a channel based in an energy channel scan.

8.1.1. Random Channel

Choosing a random channel is the easiest but not the best way. The reason is the chosen channel may be occupied by another wireless transmission. It is recommended to perform an energy channel scan to make sure the chosen channel is the best election.

If this option is used, a random value between '0x0B'-'0x1A' (XBee-PRO values are 0x0C-0x17) should be chosen. This value will be used as the input parameter in the API function responsible of setting the channel.

Example of use

```javascript
{x
    xbee802.setChannel(0x10); // Set Random Channel
}
```

8.1.2. Channel based on Energy Channel

Choosing the channel with minimum energy is the best way to choose a channel for a network. There are some parameters involved in this choice: Scan Channels and Energy Scan (see chapter 5).

The process for choosing a free channel is:

1. Change the list of channels to scan using the API function related, setting 0xFFFF to scan all the channels.
2. Perform a energy scan channel using the API function related.
3. A channel is chosen among the scanned channels, selecting the channel with minimum energy.

Example of use

```javascript
{x
    xbee802.setScanningChannels(0xFF,0xFF); // Set the list of channels to scan
    xbee802.setDurationEnergyChannels(2);//Get the maximal energy on each scanned channel
    /* Choose the minimal energy reading: xbee802.energyChannels[0-15];
    xbee802.energyChannels[0] will contain energy found in channel 0x0B and so on */
    xbee802.setChannel(0x0E); // Set Minimal Energy Channel
}
```

8.2. Choosing a PAN ID

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

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

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

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

Joining an existing network process requires some knowledge about the network to join. Three parameters are needed: channel (it may be unknown), PAN ID and security (explained in chapter 11).

9.1. Network parameters are known

When these three parameters are known, the process is reduced to set the parameters in the node.

9.1.1. Channel

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

Example of use

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

9.1.2. PAN ID

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

Example of use

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

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

9.2. Network parameters are unknown

There are some possibilities of unknowing parameters.

9.2.1. PAN ID unknown & Channel known

It is necessary to know PAN ID before joining a network. 802.15.4 doesn't support a process to discover the PAN ID if it is unknown, so it is an application level task to develop. However, a process can be executed in this case, with some previous configurations in the nodes and network.

This process is based on sending a broadcast message with 0xFFFF PAN ID on the channel the network is working. Despite not having the PAN ID, the nodes in the network will receive the message. To carry out this process, the nodes should have been configured to answer this kind of messages containing its PAN ID. Some kind of security can be applied, as sending a number that only the ‘good’ nodes know. This is a matter to be solved by the developer.

Example of use

```c
{  
  char* data;  
  panid={0xFF,0xFF}; // array containing PAN ID broadcast  
  xbee802.setPAN(panid); // Set PANID broadcast  
  packetXBee* paq_sent; // create packet to send  
  paq_sent->mode=BROADCAST; // set Broadcast mode  
  paq_sent->packetID=0x52; // set ID application level  
  paq_sent->opt=0x00; // set options. No option selected.
}
Joining an Existing Network

for(int c=0;c<50;c++) // Set the data
{
    data[c]=A;
}
xbee802.setOriginParams(paq_sent, “ASKPANID”, NI_TYPE);
xbee802.setDestinationParams(paq_sent, “000000000000FFFF”, data, MAC_TYPE, DATA_ABSOLUTE);

state=xbee802.sendXBee(paq_sent); // Call function responsible to send data

/* wait until a node answers with PANID */
state=xbee802.treatData(); // Read packet received
/* extract PAN ID sent */
panid={0x33,0x31}; // PAN ID received
xbee802.setPAN(panid); // Set PAN ID of the network

9.2.2. Channel unknown & PAN ID known

If the channel is unknown, 802.15.4 doesn’t provide a method to discover it, but at application level it can be discovered. The process is based on scanning each channel to discover some nodes or a specific node.

A method to make this process shorter is scanning the channels to start the discovery nodes at the minimum energy channels is described below.

Example of use
{
    panid={0x33,0x31}; // array containing PAN ID of the network
    xbee802.setPAN(panid); // Set PANID of the network
    xbee802.setScanningChannels(0xFF,0xFF);// list of channels to scan
    xbee802.setDurationEnergyChannels(2);// get energy on each channel
    /* energyChannel[0-15] contains each channel energy */
    /* order the array elements */
    while(xbee802.totalScannedBrothers==0)
    {
        xbee802.setChannel(0x0B); // It is supposed it is the maximum energy channel
        xbee802.scanNetwork(); // perform a Node Discovery on each channel until response
    }
    /* when exiting the loop, that will be the correct channel */
}

9.2.3. Channel & PAN ID unknown

If both parameters are unknown, it is possible to discover the parameters if some previously nodes have been configured, as in previous cases.

The application level process to follow is based on searching the energy on each channel and sending a broadcast message with PANID=0xFFFF on each channel, waiting for a response of some nodes.

Example of use
{
    panid={0xFF,0xFF}; // array containing PAN ID broadcast
    xbee802.setPAN(panid); // Set PANID broadcast
    packetXBee* paq_sent; // create packet to send
    char* data;
    paq_sent->mode=BROADCAST; // set Broadcast mode
    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;
    }
}
Joining an Existing Network

```c
xbee802.setScanningChannels(0xFF, 0xFF);// list of channels to scan
xbee802.setDurationEnergyChannels(2);// get energy on each channel
*/ energyChannel[0-15] contains each channel energy */
/* order the array elements */
while(xbee802.totalScannedBrothers==0)
{
    xbee802.setChannel(xbee802.energyChannel[0]); // Starting at first element.
    xbee802.setOriginParams(paq_sent, “ASKPANID”, NI_TYPE);
    xbee802.setDestinationParams(paq_sent,”000000000000FFFF”,data,MAC_TYPE,
    DATA_ABSOLUTE);
    xbee802.treatData(paq_sent);// Send broadcast message.
} /* when exiting the loop, that will be the correct channel and PAN ID will have been extracted */
```

9.3. Node Discovery

Once these parameters have been set, the node is part of the network, so now we can discover other nodes in the network.

Some parameters are involved in a node discovery process.

9.3.1. 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 as explained in chapter 6.

Example of use:

```c
{ 
    xbee802.scanNetwork(); // Discovery nodes
    if(xbee802.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].RSSI` → stores the RSSI of the response

9.3.2. Node Discovery Time

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

Example of use:

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

\texttt{xbee802.scanTime} → stores the time a node will wait for responses.

### 9.3.3. Node Discover Options

Enables node discover self-response on the module.

Example of use:

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

Available Information

\texttt{xbee802.discoveryOptions} → stores the selected options for ND

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

### 9.3.4. Questions related with Discovering Nodes

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

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

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

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

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

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

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

- **Node Discover Time. Tests.**

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

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

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

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

- **Increasing output power level vs Number of brothers found**
Output power level is an important parameter for scenarios in which the battery life is critical. In our tests, using the minimum output power level (-10dBm), very good results have been discovered. In indoor scenario, two nodes separated around 20 meters in different floors were discovered without any problems. In outdoor scenario, two nodes separated around 50 meters were discovered without problems.

However, it is important to try on each application scenario and choose the minimum output power level to increase the battery life to the maximum.

- **ND without knowing security key**

As explained in question 2, if a ND is performed without knowing security key, no response will be received.
10. Sleep Options

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

10.1. Sleep Mode

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

Two different options can be set:

• 1: Pin Hibernate Mode.
• 2: Pin Doze Mode.

Example of use:

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

Related Variables

`xbee802.sleepMode` → stores the sleep mode in a module

10.2. Pin Hibernate Mode

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

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

Example of use:

```c
{xbee802.setSleepMode(1); // Set Sleep Mode to Hibernate Mode
 xbee802.sleep(); // Set XBee to sleep
delay(5000); // wait 5 seconds
 xbee802.wake(); // Wake up the XBee
}
```

10.3. Pin Doze Mode

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

Example of use:

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

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

Example of use:

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

11. Security and Data Encryption

11.1. IEEE 802.15.4 Security and Data encryption Overview

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

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

![IEEE 802.15.4 Frame](image)

**Figure 11.1 : IEEE 802.15.4 Frame**

11.2. Security in API libraries

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

11.2.1. Encryption Enable

 Enables the 128-bit AES encryption in the modules. When encryption is enabled, the module will always use its 64-bit address as the source address for RF packets. With encryption enabled and a 16-bit address set, receiving modules will only be able to issue receive 64-bit indicators.

Example of use:

```
{ 
    xbee802.encryptionMode(1);  // Enable encryption mode
}
```
Security and Data Encryption

Related Variables

\texttt{xbee802.encryptMode} \rightarrow \text{stores if security is enabled or not}

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.

![AES-CTR Encryption Frame](image)

**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

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

Related Variables

\texttt{xbee802.linkKey} \rightarrow \text{stores the key that has been set in the network}

**11.2.3. Maximum Data Payloads**

As explained in chapter 3, maximum data payloads are decreased when using cyphering.

<table>
<thead>
<tr>
<th></th>
<th>@16bit Unicast</th>
<th>@64bit Unicast</th>
<th>Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypted</td>
<td>98Bytes</td>
<td>94Bytes</td>
<td>95Bytes</td>
</tr>
<tr>
<td>Un-Encrypted</td>
<td>100Bytes</td>
<td>100Bytes</td>
<td>100Bytes</td>
</tr>
</tbody>
</table>

![Maximum Payloads](image)
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.

As explained previously, security prevents a network from being attacked by an stranger node. Node Discovery process will not work without knowing security key.

11.3.1. Creating a network enabling security

When creating a network, as explained in chapter 7, security measures can be set. After choosing PAN ID and channel, to enable security a key should be chosen. This key is a 128-bit AES encryption key that has to be the same in all the nodes of the same network.

Example of use:

```cpp
{ 
    panid={0x33,0x31};
    xbee802.setChannel(0x10); // Set Channel
    xbee802.setPAN(panid); // Set PAN ID
    char* KEY="WaspmoteLinkKey!"
    xbee802.encryptionMode(1); // Enable encryption mode
    xbee802.setLinkKey(KEY); // Set Link Key
}
```

Available Information

- `xbee802.channel` → it will store the channel selected
- `xbee802.PAN_ID` → it will store the PAN ID selected
- `xbee802.encryptMode` → it will store if security is enabled
- `xbee802.linkKey` → it will store the selected key

11.3.2. Joining a security enabled network

When joining a network with security enabled, PAN ID and channel has to be set in the node, as explained in chapter 8, but security parameters have to be enabled for a full communication. The joining node should know the 128-bit AES encryption key and enable security setting this key.

Example of use:

```cpp
{ 
    panid={0x33,0x31};
    xbee802.setChannel(0x10); // Set Channel
    xbee802.setPAN(panid); // Set PAN ID
    char* KEY="WaspmoteLinkKey!"
    xbee802.encryptionMode(1); // Enable encryption mode
    xbee802.setLinkKey(KEY); // Set Link Key
}
```

Available Information

- `xbee802.channel` → it will store the channel selected
- `xbee802.PAN_ID` → it will store the PAN ID selected
- `xbee802.encryptMode` → it will store if security is enabled
- `xbee802.linkKey` → it will store the selected key
12. Code examples and extended information

For more information about the Wasp mote hardware platform go to:

http://www.libelium.com/wasp mote
http://www.libelium.com/support/wasp mote
http://www.libelium.com/development/wasp mote