# Getting Started

## 1 Requirements

## 2 Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Getting Started</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>User Documentation</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Examples</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>FAQ</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>API reference</td>
<td>6</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Get started with XBee Python library</td>
<td>6</td>
</tr>
<tr>
<td>2.5.2</td>
<td>XBee terminology</td>
<td>14</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Work with XBee classes</td>
<td>18</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Configure the XBee device</td>
<td>24</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Discover the XBee network</td>
<td>35</td>
</tr>
<tr>
<td>2.5.1.1</td>
<td>Install your software</td>
<td>6</td>
</tr>
<tr>
<td>2.5.1.2</td>
<td>Configure your XBee modules</td>
<td>8</td>
</tr>
<tr>
<td>2.5.1.3</td>
<td>Run your first XBee Python application</td>
<td>11</td>
</tr>
<tr>
<td>2.5.2.1</td>
<td>RF modules</td>
<td>14</td>
</tr>
<tr>
<td>2.5.2.2</td>
<td>XBee RF modules</td>
<td>14</td>
</tr>
<tr>
<td>2.5.2.3</td>
<td>Radio firmware</td>
<td>14</td>
</tr>
<tr>
<td>2.5.2.4</td>
<td>Radio communication protocols</td>
<td>14</td>
</tr>
<tr>
<td>2.5.2.5</td>
<td>Radio module operating modes</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2.6</td>
<td>API frames</td>
<td>17</td>
</tr>
<tr>
<td>2.5.2.7</td>
<td>AT settings or commands</td>
<td>17</td>
</tr>
<tr>
<td>2.5.3.1</td>
<td>Instantiate an XBee device</td>
<td>19</td>
</tr>
<tr>
<td>2.5.3.2</td>
<td>Open the XBee device connection</td>
<td>20</td>
</tr>
<tr>
<td>2.5.3.3</td>
<td>Close the XBee device connection</td>
<td>23</td>
</tr>
<tr>
<td>2.5.4.1</td>
<td>Read and set common parameters</td>
<td>24</td>
</tr>
<tr>
<td>2.5.4.2</td>
<td>Read, set and execute other parameters</td>
<td>26</td>
</tr>
<tr>
<td>2.5.4.3</td>
<td>Apply configuration changes</td>
<td>29</td>
</tr>
<tr>
<td>2.5.4.4</td>
<td>Write configuration changes</td>
<td>30</td>
</tr>
<tr>
<td>2.5.4.5</td>
<td>Reset the device</td>
<td>30</td>
</tr>
<tr>
<td>2.5.4.6</td>
<td>Configure Wi-Fi settings</td>
<td>31</td>
</tr>
<tr>
<td>2.5.4.7</td>
<td>Configure Bluetooth settings</td>
<td>33</td>
</tr>
<tr>
<td>2.5.5.1</td>
<td>Configure the discovery process</td>
<td>36</td>
</tr>
<tr>
<td>2.5.5.2</td>
<td>Discover the network</td>
<td>37</td>
</tr>
<tr>
<td>2.5.5.3</td>
<td>Access the discovered devices</td>
<td>40</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>2.5.4 Add and remove devices manually</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>2.5.5 Listen to network modification events</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>2.5.6 Communicate with XBee devices</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>2.5.6.1 Send and receive data</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>2.5.6.2 Send and receive explicit data</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>2.5.6.3 Send and receive IP data</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>2.5.6.4 Send and receive SMS messages</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>2.5.6.5 Send and receive Bluetooth data</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>2.5.6.6 Send and receive MicroPython data</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>2.5.6.7 Receive modem status events</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>2.5.6.8 Communicate using XBee sockets</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>2.5.7 Handle analog and digital IO lines</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>2.5.7.1 Configure the IO lines</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>2.5.7.2 Read IO samples</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>2.5.7.3 Change detection sampling</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>2.5.8 Update the XBee</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>2.5.8.1 Update the XBee firmware</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>2.5.8.2 Update the XBee file system</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>2.5.8.3 Apply an XBee profile</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>2.5.9 Log events</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>2.5.9.1 Logging level</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2.5.10 XBee Python samples</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>2.5.10.1 Configuration samples</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>2.5.10.2 Network samples</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>2.5.10.3 Communication samples</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>2.5.10.4 IO samples</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>2.5.10.5 Firmware samples</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>2.5.10.6 File system samples</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>2.5.10.7 Profile samples</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>2.5.11 Frequently Asked Questions (FAQs)</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>2.5.11.1 What is XCTU and how do I download it?</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>2.5.11.2 How do I find the serial port and baud rate of my module?</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>2.5.11.3 Can I use the XBee Python Library with modules in AT operating mode?</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>2.5.11.4 I get the Python error ImportError: No module named 'serial'</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>2.5.11.5 I get the Python error ImportError: No module named 'arp'</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>2.5.12 API reference</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>2.5.12.1 digi package</td>
<td>114</td>
<td></td>
</tr>
</tbody>
</table>

3 Indices and tables
4 License

Python Module Index

Index
XBee devices allow you to enable wireless connectivity to your projects creating a network of connected devices. They provide features to exchange data with other devices in the network, configure them and control their I/O lines. An application running in an intelligent device can take advantage of these features to monitor and manage the entire network.

Despite the available documentation and configuration tools for working with XBee devices, it is not always easy to develop these kinds of applications.

The XBee Python Library is a Python API that dramatically reduces the time to market of XBee projects developed in Python and facilitates the development of these types of applications, making it an easy and smooth process. The XBee Python Library includes the following features:

- Support for multiple XBee devices and protocols.
- High abstraction layer provides an easy-to-use workflow.
- Ability to configure local and remote XBee devices of the network.
- Discovery feature finds remote nodes on the same network as the local module.
- Ability to transmit and receive data from any XBee device on the network.
- Ability to manage the General Purpose Input and Output lines of all your XBee devices.
- Ability to send and receive data from other XBee interfaces (Serial, Bluetooth Low Energy and MicroPython).

This portal provides the following documentation to help you with the different development stages of your Python applications using the XBee Python Library.
The XBee Python library requires the following components in order to work properly:

- **Python 3.** You can get it from https://www.python.org/getit/

- **PySerial 3.** Install it with pip (`pip install pyserial`) or refer to the PySerial installation guide for further information about getting PySerial.

- **SRP** Install it with pip (`pip install srp`).
The XBee Python library documentation is split in different sections:

- *Getting Started*
- *User Documentation*
- *Examples*
- *FAQ*
- *API reference*

## 2.1 Getting Started

Perform your first steps with the XBee Python library. Learn how to setup your environment and communicate with your XBee devices using the library.

- *Get started with XBee Python library*

## 2.2 User Documentation

Access detailed information about the different features and capabilities provided by the library and how to use them.

- *XBee terminology*
- *Work with XBee classes*
- *Configure the XBee device*
- *Discover the XBee network*
- *Communicate with XBee devices*
- *Handle analog and digital IO lines*
• Update the XBee
• Log events

2.3 Examples

The library includes a good amount of examples that demonstrate most of the functionality that it provides.

• XBee Python samples

2.4 FAQ

Find the answer to the most common questions or problems related to the XBee Python library in the FAQ section.

• Frequently Asked Questions (FAQs)

2.5 API reference

The API reference contains more detailed documentation about the API for developers who are interested in using and extending the library functionality.

• API reference

2.5.1 Get started with XBee Python library

This getting started guide describes how to set up your environment and use the XBee Python Library to communicate with your XBee devices. It explains how to configure your modules and write your first XBee Python application.

The guide is split into 3 main sections:

• Install your software
• Configure your XBee modules
• Run your first XBee Python application

2.5.1.1 Install your software

The following software components are required to write and run your first XBee Python application:

• Python 3
• PySerial 3
• SRP
• XBee Python library software
• XCTU
Python 3

The XBee Python library requires Python 3. If you don’t have Python 3, you can get it from https://www.python.org/getit/.

**Warning:** The XBee Python library is currently only compatible with Python 3.

PySerial 3

You must be able to communicate with the radio modules over a serial connection. The XBee Python library uses the **PySerial** module for that functionality.

This module is automatically downloaded when you install the XBee Python library.

SRP

The XBee Python library uses the **SRP** module to authenticate with XBee devices over Bluetooth Low Energy.

This module is automatically downloaded when you install the XBee Python library.

XBee Python library software

The best way to install the XBee Python library is with the **pip** tool (which is what Python uses to install packages). The pip tool comes with recent versions of Python.

To install the library, run this command in your terminal application:

```
$ pip install digi-xbee
```

The library is automatically downloaded and installed in your Python interpreter.

Get the source code

The XBee Python library is actively developed on GitHub, where the code is always available. You can clone the repository with:

```
$ git clone git@github.com:digidotcom/xbee-python.git
```

XCTU

XCTU is a free multi-platform application that enables developers to interact with Digi RF modules through a simple-to-use graphical interface. It includes new tools that make it easy to set up, configure, and test XBee RF modules.

For instructions on downloading and using XCTU, go to:

http://www.digi.com/xctu

Once you have downloaded XCTU, run the installer and follow the steps to finish the installation process.

After you load XCTU, a message about software updates appears. We recommend you always update XCTU to the latest available version.

2.5. API reference
2.5.1.2 Configure your XBee modules

You need to configure two XBee devices. One module (the sender) sends “Hello XBee World!” using the Python application. The other device (the receiver) receives the message.

To communicate, both devices must be working in the same protocol (802.15.4, ZigBee, DigiMesh, Point-to-Multipoint, or Wi-Fi) and must be configured to operate in the same network.

Note: If you are getting started with cellular, you only need to configure one device. Cellular protocol devices are connected directly to the Internet, so there is no network of remote devices to communicate with them. For the cellular protocol, the XBee application demonstrated in the getting started guide differs from other protocols. The cellular protocol sends and reads data from an echo server.

Use XCTU to configure the devices. Plug the devices into the XBee adapters and connect them to your computer’s USB or serial ports.

Note: For more information about XCTU, see the XCTU User Guide. You can also access the documentation from the Help menu of the tool.

Once XCTU is running, add your devices to the tool and then select them from the Radio Modules section. When XCTU is finished reading the device parameters, complete the following steps according to your device type. Repeat these steps to configure your XBee devices using XCTU.

- **802.15.4 devices**
- **ZigBee devices**
- **DigiMesh devices**
- **DigiPoint devices**
- **Cellular devices**
- **Wi-Fi devices**

### 802.15.4 devices

1. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.

2. Make sure API mode (API1 or API2) is enabled. To do so, set the AP parameter value to 1 (API mode without escapes) or 2 (API mode with escapes).

3. Configure **ID** (PAN ID) setting to CAFE.

4. Configure **CH** (Channel setting) to C.

5. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.

6. Once you have configured both modules, check to make sure they can see each other. Click Discover radio modules in the same network, the second button of the device panel in the Radio Modules view. The other device must be listed in the Discovering remote devices dialog.

Note: If the other module is not listed, reboot both devices by pressing the Reset button of the carrier board and try adding the device again. If the list is still empty, see the product manual for your device.
ZigBee devices

1. For old ZigBee devices (S2 and S2B), make sure the devices are using API firmware. The firmware appears in the **Function** label of the device in the Radio Modules view.
   - One of the devices must be a coordinator - Function: ZigBee Coordinator API
   - Digi recommends the other one is a router - Function: ZigBee Router AP.

   **Note:** If any of the two previous conditions is not satisfied, you must change the firmware of the device. Click the **Update firmware** button of the Radio Configuration toolbar.

2. Click **Load default firmware settings** in the **Radio Configuration** toolbar to load the default values for the device firmware.

3. Do the following:
   - If the device has the **AP** parameter, set it to 1 (API mode without escapes) or 2 (API mode with escapes).
   - If the device has the **CE** parameter, set it to **Enabled** in the coordinator.

4. Configure **ID** (PAN ID) setting to **C001BEE**.

5. Configure **SC** (Scan Channels) setting to **FFF**.

6. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.

7. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

   **Note:** If the other module is not listed, reboot both devices by pressing the **Reset** button of the carrier board and try adding the device again. If the list is still empty, go to the corresponding product manual for your devices.

DigiMesh devices

1. Click **Load default firmware settings** in the **Radio Configuration** toolbar to load the default values for the device firmware.

2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be 1 (API mode without escapes) or 2 (API mode with escapes).

3. Configure **ID** (PAN ID) setting to **CAFE**.

4. Configure **CH** (Operating Channel) to **C**.

5. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.

6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

   **Note:** If the other module is not listed, reboot both devices by pressing the **Reset** button of the carrier board and try adding the device again. If the list is still empty, go to the corresponding product manual for your devices.
DigiPoint devices

1. Click **Load default firmware settings** in the Radio Configuration toolbar to load the default values for the device firmware.

2. Ensure the API mode (API1 or API2) is enabled. To do so, the AP parameter value must be 1 (API mode without escapes) or 2 (API mode with escapes).

3. Configure **ID** (PAN ID) setting to **CAFE**.

4. Configure **HP** (Hopping Channel) to **5**.

5. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.

6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the Radio Modules view. The other device must be listed in the Discovering remote devices dialog.

**Note:** If the other module is not listed, reboot both devices by pressing the **Reset** button of the carrier board and try adding the device again. If the list is still empty, go to the corresponding product manual for your devices.

Cellular devices

1. Click **Load default firmware** settings in the Radio Configuration toolbar to load the default values for the device firmware.

2. Ensure the API mode (API1 or API2) is enabled. To do so, the AP parameter value must be 1 (API mode without escapes) or 2 (API mode with escapes).

3. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.

4. Verify the module is correctly registered and connected to the Internet. To do so check that the LED on the development board blinks. If it is solid or has a double-blink, registration has not occurred properly. Registration can take several minutes.

**Note:** In addition to the LED confirmation, you can check the IP address assigned to the module by reading the **MY** parameter and verifying it has a value different than **0.0.0.0**.

Wi-Fi devices

1. Click **Load default firmware** settings in the Radio Configuration toolbar to load the default values for the device firmware.

2. Ensure the API mode (API1 or API2) is enabled. To do so, the AP parameter value must be 1 (API mode without escapes) or 2 (API mode with escapes).

3. Connect to an access point:
   1. Click the **Active Scan** button.
   2. Select the desired access point from the list of the **Active Scan** result dialog.
   3. If the access point requires a password, type your password.
   4. Click the **Connect** button and wait for the module to connect to the access point.
4. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.

5. Verify the module is correctly connected to the access point by checking the IP address assigned to the module by reading the **MY** parameter and verifying it has a value different than **0.0.0.0**.

### 2.5.1.3 Run your first XBee Python application

The XBee Python application demonstrated in the guide broadcasts the message *Hello XBee World!* from one of the devices connected to your computer (the sender) to all remote devices on the same network as the sender. Once the message is sent, the receiver XBee module must receive it. You can use XCTU to verify receipt.

The commands to be executed depend on the protocol of the XBee devices. Follow the corresponding steps depending on the protocol of your XBee devices.

- **ZigBee, DigiMesh, DigiPoint or 802.15.4 devices**
- **Wi-Fi devices**
- **Cellular devices**

#### ZigBee, DigiMesh, DigiPoint or 802.15.4 devices

Follow these steps to send the broadcast message and verify that it is received successfully:

1. First, prepare the **receiver** XBee device in XCTU to verify that the broadcast message sent by the **sender** device is received successfully. Follow these steps to do so:
   1. Launch XCTU.
   2. Add the **receiver** module to XCTU.
   3. Click **Open the serial connection with the radio module** to switch to **Consoles working mode** and open the serial connection. This allows you to see the data when it is received.

2. Open the Python interpreter and write the application commands.
   1. Import the **XBeeDevice** class by executing the following command:

   ```python
   >>> from digi.xbee.devices import XBeeDevice
   ```

   2. Instantiate a generic XBee device:

   ```python
   >>> device = XBeeDevice("COM1", 9600)
   ```

   **Note:** Remember to replace the COM port with the one your **sender** XBee device is connected to. In UNIX-based systems, the port usually starts with /dev/tty.

   3. Open the connection with the device:

   ```python
   >>> device.open()
   ```

   4. Send the *Hello XBee World!* broadcast message.

   ```python
   >>> device.send_data_broadcast("Hello XBee World!")
   ```

   5. Close the connection with the device:
3. Verify that the message is received by the receiver XBee in XCTU. An RX (Receive) frame should be displayed in the Console log with the following information:

<table>
<thead>
<tr>
<th>Start delimiter</th>
<th>7E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Depends on the XBee protocol</td>
</tr>
<tr>
<td>Frame type</td>
<td>Depends on the XBee protocol</td>
</tr>
<tr>
<td>16/64-bit source address</td>
<td>XBee sender’s 16/64-bit address</td>
</tr>
<tr>
<td>Options</td>
<td>02</td>
</tr>
<tr>
<td>RF data/Received data</td>
<td>48 65 6C 6C 6F 20 58 42 65 65 20 57 6F 72 6C 64 21</td>
</tr>
</tbody>
</table>

**Wi-Fi devices**

Wi-Fi devices send broadcast data using the `send_ip_data_broadcast()` command instead of the `send_data_broadcast()` one. For that reason, you must instantiate a `WiFiDevice` instead of a generic `XBeeDevice` to execute the proper command.

Follow these steps to send the broadcast message and verify that it is received successfully:

1. First, prepare the receiver XBee device in XCTU to verify that the broadcast message sent by the sender device is received successfully by the receiver device.
   1. Launch XCTU.
   2. Add the receiver module to XCTU.
   3. Click **Open the serial connection with the radio module** to switch to **Consoles working mode** and open the serial connection. This allows you to see the data when it is received.

2. Open the Python interpreter and write the application commands.
   1. Import the `WiFiDevice` class by executing the following command:

      ```python
      > from digi.xbee.devices import WiFiDevice
      ```

   2. Instantiate a Wi-Fi XBee device:

      ```python
      > device = WiFiDevice("COM1", 9600)
      ```

   **Note:** Remember to replace the COM port with the one your sender XBee device is connected to. In UNIX-based systems, the port usually starts with `/dev/tty`.

3. Open the connection with the device:

   ```python
   > device.open()
   ```

4. Send the *Hello XBee World!* broadcast message.

   ```python
   > device.send_ip_data_broadcast(9750, "Hello XBee World!")
   ```

5. Close the connection with the device:

   ```python
   > device.close()
   ```
3. Verify that the message is received by the receiver XBee in XCTU. An **RX IPv4 frame** should be displayed in the **Console log** with the following information:

<table>
<thead>
<tr>
<th>Start delimiter</th>
<th>7E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>00 1C</td>
</tr>
<tr>
<td>Frame type</td>
<td>B0</td>
</tr>
<tr>
<td>IPv4 source address</td>
<td>XBee Wi-Fi sender’s IP address</td>
</tr>
<tr>
<td>16-bit dest port</td>
<td>26 16</td>
</tr>
<tr>
<td>16-bit source port</td>
<td>26 16</td>
</tr>
<tr>
<td>Protocol</td>
<td>00</td>
</tr>
<tr>
<td>Status</td>
<td>00</td>
</tr>
<tr>
<td>RF data</td>
<td>48 65 6C 6C 6F 20 58 42 65 65 20 57 6F 72 6C 64 21</td>
</tr>
</tbody>
</table>

**Cellular devices**

Cellular devices are connected directly to the Internet, so there is no network of remote devices to communicate with them. For cellular protocol, the application demonstrated in this guide differs from other protocols.

The application sends and reads data from an echo server. Follow these steps to execute it:

1. Open the Python interpreter and write the application commands.

   1. Import the **CellularDevice**, **IPProtocol** and **IPv4Address** classes:

   ```python
   from digi.xbee.devices import CellularDevice
   from digi.xbee.models.protocol import IPProtocol
   from ipaddress import IPv4Address
   ```

   2. Instantiate a cellular XBee device:

   ```python
   device = CellularDevice("COM1", 9600)
   ```

   **Note:** Remember to replace the COM port by the one your Cellular XBee device is connected to. In UNIX-based systems, the port usually starts with `/dev/tty`.

   3. Open the connection with the device:

   ```python
   device.open()
   ```

   4. Send the **Hello XBee World!** message to the echo server with IP 52.43.121.77 and port 11001 using the **TCP IP** protocol.

   ```python
   device.send_ip_data(IPv4Address("52.43.121.77"), 11001, IPProtocol.TCP, "Hello XBee World!")
   ```

   5. Read and print the response from the echo server. If response cannot be received, print **ERROR**.

   ```python
   ip_message = device.read_ip_data()
   print(ip_message.data.decode("utf8") if ip_message is not None else "ERROR")
   ```

   6. Close the connection with the device:

   ```python
   device.close()
   ```
2.5.2 XBee terminology

This section covers basic XBee concepts and terminology. The XBee Python library manual refers to these concepts frequently, so it is important to understand these concepts.

2.5.2.1 RF modules

A radio frequency (RF) module is a small electronic circuit used to transmit and receive radio signals on different frequencies. Digi produces a wide variety of RF modules to meet the requirements of almost any wireless solution, such as long-range, low-cost, and low power modules.

2.5.2.2 XBee RF modules

XBee is the brand name of a family of RF modules produced by Digi International Inc. XBee RF modules are modular products that make it easy and cost-effective to deploy wireless technology. Multiple protocols and RF features are available, giving customers enormous flexibility to choose the best technology for their needs.

The XBee RF modules are available in two form factors: Through-Hole and Surface Mount, with different antenna options. Almost all modules are available in the Through-Hole form factor and share the same footprint.

2.5.2.3 Radio firmware

Radio firmware is the program code stored in the radio module’s persistent memory that provides the control program for the device. From the local web interface of the XBee Gateway, you can update or change the firmware of the local XBee module or any other module connected to the same network. This is a common task when changing the role of the device or updating to the latest version of the firmware.

2.5.2.4 Radio communication protocols

A radio communication protocol is a set of rules for data exchange between radio devices. An XBee module supports a specific radio communication protocol depending on the module and its radio firmware.

Following is the complete list of protocols supported by the XBee radio modules:

- IEEE 802.15.4
- ZigBee
- ZigBee Smart Energy
- DigiMesh (Digi proprietary)
- ZNet
- IEEE 802.11 (Wi-Fi)
• Point-to-multipoint (Digi proprietary)
• XSC (XStream compatibility)
• Cellular
• Thread

Note: Not all XBee devices can run all these communication protocols. The combination of XBee hardware and radio firmware determines the protocol that an XBee device can execute. Refer to the XBee RF Family Comparison Matrix for more information about the available XBee RF modules and the protocols they support.

2.5.2.5 Radio module operating modes

The operating mode of an XBee radio module establishes the way a user, or any microcontroller attached to the XBee, communicates with the module through the Universal Asynchronous Receiver/Transmitter (UART) or serial interface. Depending on the firmware and its configuration, the radio modules can work in three different operating modes:

• Application Transparent (AT) operating mode
• API operating mode
• API escaped operating mode

In some cases, the operating mode of a radio module is established by the firmware version and the firmware’s AP setting. The module’s firmware version determines whether the operating mode is AT or API. The firmware’s AP setting determines if the API mode is escaped (AP = 2) or not (AP = 1). In other cases, the operating mode is only determined by the AP setting, which allows you to configure the mode to be AT (AP = 0), API (AP = 1) or API escaped (AP = 2).

Application Transparent (AT) operating mode

In Application Transparent (AT) or transparent operating mode, all serial data received by the radio module is queued up for RF transmission. When the module receives RF data, it sends the data out through the serial interface.
To configure an XBee module operating in AT, put the device in command mode to send the configuration commands.

**Command mode**

When the radio module is working in AT operating mode, configure settings using the command mode interface.

To enter command mode, send the 3-character command sequence through the serial interface of the radio module, usually `+++`, within one second. Once the command mode has been established, the module sends the reply `OK`, the command mode timer starts, and the radio module can receive AT commands.

The structure of an AT command follows this format:

```
AT[ASCII command][Space (optional)][Parameter (optional)][Carriage return]
```

Example:

```
ATNI MyDevice\r
```

If no valid AT commands are received within the command mode timeout, the radio module automatically exits command mode. You can also exit command mode issuing the `CN` command (Exit Command mode).

**API operating mode**

Application Programming Interface (API) operating mode is an alternative to AT operating mode. API operating mode requires that communication with the module through a structured interface; that is, data communicated in API frames.

The API specifies how commands, command responses, the module sends and receives status messages using the serial interface. API operation mode enables many operations, such as the following:

- Configure the XBee device itself.
- Configure remote devices in the network.
- Manage data transmission to multiple destinations.
- Receive success/failure status of each transmitted RF packet.
- Identify the source address of each received packet.

Depending on the AP parameter value, the device can operate in one of two modes: API (AP = 1) or API escaped (AP = 2) operating mode.

**API escaped operating mode**

API escaped operating mode (AP = 2) works similarly to API mode. The only difference is that when working in API escaped mode, some bytes of the API frame specific data must be escaped.

Use API escaped operating mode to add reliability to the RF transmission, which prevents conflicts with special characters such as the start-of-frame byte (0x7E). Since 0x7E can only appear at the start of an API packet, if 0x7E is received at any time, you can assume that a new packet has started regardless of length. In API escaped mode, those special bytes are escaped.

**Escape characters**

When sending or receiving an API frame in API escaped mode, you must escape (flag) specific data values so they do not interfere with the data frame sequence. To escape a data byte, insert 0x7D and follow it with the byte being escaped, XOR’d with 0x20.
The following data bytes must be escaped:
- 0x7E: Frame delimiter
- 0x7D: Escape
- 0x11: XON
- 0x13: XOFF

### 2.5.2.6 API frames

An API frame is the structured data sent and received through the serial interface of the radio module when it is configured in API or API escaped operating modes. API frames are used to communicate with the module or with other modules in the network.

An API frame has the following structure:

<table>
<thead>
<tr>
<th>Start Delimiter (Byte 1)</th>
<th>Length (Bytes 2-3)</th>
<th>Frame Data (Bytes 4-n)</th>
<th>Checksum (Byte n+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7E</td>
<td>MSB</td>
<td>LSB</td>
<td>API-specific Structure</td>
</tr>
</tbody>
</table>

In API escaped mode, some bytes in the Length, Frame Data and Checksum fields must be escaped.

### 2.5.2.7 AT settings or commands

The firmware running in the XBee RF modules contains a group of settings and commands that you can configure to change the behavior of the module or to perform any related action. Depending on the protocol, the number of settings and meanings vary, but all the XBee RF modules can be configured with AT commands.

All the firmware settings or commands are identified with two ASCII characters and some applications and documents refer to them as **AT settings** or **AT commands**.

The configuration process of the AT settings varies depending on the operating mode of the XBee RF module.
• **AT operating mode.** In this mode, you must put the module in a special mode called command mode, so it can receive AT commands. For more information about configuring XBee RF modules working in AT operating mode, see *Application Transparent (AT) operating mode*.

• **API operating mode.** To configure or execute AT commands when the XBee RF module operates in API mode, you must generate an AT command API frame containing the AT setting identifier and the value of that setting, and send it to the XBee RF module. For more information about API frames, see *API frames*.

### 2.5.3 Work with XBee classes

When working with the XBee Python Library, start with an XBee device object that represents a physical module. A physical XBee device is the combination of hardware and firmware. Depending on that combination, the device runs a specific wireless communication protocol such as ZigBee, 802.15.4, DigiMesh, Wi-Fi, or cellular. An XBeeDevice class represents the XBee module in the API.

Most of the protocols share the same features and settings, but there are some differences between them. For that reason, the XBee Python Library also includes a set of classes that represent XBee devices running different communication protocols. The XBee Python Library supports one XBee device class per protocol, as follows:

- XBee ZigBee device (*ZigBeeDevice*)
- XBee 802.15.4 device (*Raw802Device*)
- XBee DigiMesh device (*DigiMeshDevice*)
- XBee Point-to-multipoint device (*DigiPointDevice*)
- XBee IP devices (This is a non-instantiable class)
  - XBee Cellular device (*CellularDevice*)
  - XBee Wi-Fi device (*WiFiDevice*)

All these XBee device classes allow you to configure the physical XBee device, communicate with the device, send data to other nodes on the network, receive data from remote devices, and so on. Depending on the class, you may have additional methods to execute protocol-specific features or similar methods.

To work with the API and perform actions involving the physical device, you must instantiate a generic *XBeeDevice* object or one that is protocol-specific. This documentation refers to the *XBeeDevice* object generically when describing the different features, but they can be applicable to any XBee device class.
2.5.3.1 Instantiate an XBee device

When you are working with the XBee Python Library, the first step is to instantiate an XBee device object. The API works well using the generic XBeeDevice class, but you can also instantiate a protocol-specific XBee device object if you know the protocol your physical XBee device is running.

An XBee device is represented as either local or remote in the XBee Python Library, depending upon how you communicate with the device.

Local XBee device

A local XBee device is the object in the library representing the device that is physically attached to your PC through a serial or USB port. The classes you can instantiate to represent a local device are listed in the following table:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBeeDevice</td>
<td>Generic object, protocol-independent</td>
</tr>
<tr>
<td>ZigBeeDevice</td>
<td>ZigBee protocol</td>
</tr>
<tr>
<td>Raw802Device</td>
<td>802.15.4 protocol</td>
</tr>
<tr>
<td>DigiMeshDevice</td>
<td>DigiMesh protocol</td>
</tr>
<tr>
<td>DigiPointDevice</td>
<td>Point-to-multipoint protocol</td>
</tr>
<tr>
<td>CellularDevice</td>
<td>Cellular protocol</td>
</tr>
<tr>
<td>WiFiDevice</td>
<td>Wi-Fi protocol</td>
</tr>
</tbody>
</table>

To instantiate a generic or protocol-specific XBee device, you need to provide the following two parameters:

- Serial port name
- Serial port baud rate

**Instantiate a local XBee device**

```python
[...]
xbee = XBeeDevice("COM1", 9600)
[...
```

Remote XBee device

Remote XBee device objects represent remote nodes of the network. These are XBee devices that are not attached to your PC but operate in the same network as the attached (local) device.

**Warning:** When working with remote XBee devices, it is very important to understand that you cannot communicate directly with them. You need to provide a local XBee device operating in the same network that acts as bridge between your serial port and the remote node.

Managing remote devices is similar to managing local devices, but with limitations. You can configure them, handle their IO lines, and so on, in the same way you manage local devices. Local XBee devices have several methods for sending data to remote devices, but the remote devices cannot use these methods because they are already remote. Therefore, a remote device cannot send data to another remote device.

In the local XBee device instantiation, you can choose between instantiating a generic remote XBee device object or a protocol-specific remote XBee device. The following table lists the remote XBee device classes:
## RemoteXBeeDevice

Generic object, protocol independent

## RemoteZigBeeDevice

ZigBee protocol

## RemoteRaw802Device

802.15.4 protocol

## RemoteDigiMeshDevice

DigiMesh protocol

## RemoteDigiPointDevice

Point-to-multipoint protocol

---

**Note:** XBee Cellular and Wi-Fi protocols do not support remote devices.

To instantiate a remote XBee device object, you need to provide the following parameters:

- Local XBee device attached to your PC that serves as the communication interface.
- 64-bit address of the remote device.

`RemoteRaw802Device` objects can be also instantiated by providing the local XBee device attached to your PC and the **16-bit address** of the remote device.

**Instantiate a remote XBee device**

```python
local_xbee = XBeeDevice("COM1", 9600)
remote_xbee = RemoteXBeeDevice(local_xbee, XBee64BitAddress.from_hex_string("0013A20012345678"))
```

The local device must also be the same protocol for protocol-specific remote XBee devices.

### 2.5.3.2 Open the XBee device connection

Before trying to communicate with the local XBee device attached to your PC, you need to open its communication interface, which is typically a serial/USB port. Use the `open()` method of the instantiated XBee device, and you can then communicate and configure the device.

Remote XBee devices do not have an open method. They use a local XBee device as the connection interface. If you want to perform any operation with a remote XBee device you must open the connection of the associated local device.

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)

# Open the device connection.
local_xbee.open()
```

The `open()` method may fail for the following reasons:

- All the possible errors are caught as `XBeeException`:
  - If there is any problem with the communication, throwing a `TimeoutException`.

---

20 Chapter 2. Contents
– If the operating mode of the device is not API or API_ESCAPE, throwing an InvalidOperatingModeException.
– There is an error writing to the XBee interface, or device is closed, throwing a generic XBeeException.

The open() action performs some other operations apart from opening the connection interface of the device. It reads the device information (reads some sensitive data from it) and determines the operating mode of the device.

Use force_settings=True as open() method parameter, to reconfigure the XBee serial settings (baud rate, data bits, stop bits, etc.) to those specified in the XBee object constructor.

```python
...
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)

# Open the connection using constructor parameters: 9600 8N1.
# This reconfigures the XBee if its serial settings do not match.
local_xbee.open(force_settings=True)

[...]
```

Example: Recover XBee serial communication

The XBee Python Library includes a sample application that displays how to recover the serial connection with a local XBee. It can be located in the following path:
```
examples/configuration/RecoverSerialConnection/RecoverSerialConnection.py
```

Read device information

The read device information process reads the following parameters from the local or remote XBee device and stores them inside. You can then access parameters at any time, calling their corresponding getters.

- 64-bit address
- 16-bit address
- Node identifier
- Firmware version
- Hardware version
- IPv4 address (only for cellular and Wi-Fi modules)
- IMEI (only for cellular modules)

The read process is performed automatically in local XBee devices when opening them with the open() method. If remote XBee devices cannot be opened, you must use read_device_info() to read their device information.

Initialize a remote XBee device

```python
[...]
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Instantiate a remote XBee device object.
remote_xbee = RemoteXBeeDevice(local_xbee, XBee64BitAddress.from_hex_string("0013A20040XXXXXX"))
```

(continues on next page)
# Read the device information of the remote XBee device.
remote_xbee.read_device_info()

[...]

The `read_device_info()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - If the operating mode of the device is not `API` or `API_ESCAPE`, throwing an `InvalidOperatingModeException`.
  - If the response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, or device is closed, throwing a generic `XBeeException`.

Note: Although the `readDeviceInfo` method is executed automatically in local XBee devices when they are open, you can issue it at any time to refresh the information of the device.

Get device information

[...]

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Get the 64-bit address of the device.
addr_64 = device.get_64bit_addr()
# Get the node identifier of the device.
ode_id = device.get_node_id()
# Get the hardware version of the device.
hardware_version = device.get_hardware_version()
# Get the firmware version of the device.
firmware_version = device.get_firmware_version()
```

The read device information process also determines the communication protocol of the local or remote XBee device object. This is typically something you need to know beforehand if you are not using the generic `XBeeDevice` object.

However, the API performs this operation to ensure that the class you instantiated is the correct one. So, if you instantiated a ZigBee device and the `open()` process realizes that the physical device is actually a DigiMesh device, you receive an `XBeeDeviceException` indicating the device mismatch.

You can retrieve the protocol of the XBee device from the object executing the corresponding getter.

Get the XBee protocol

[...]

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()
```
# Get the protocol of the device.
protocol = local_xbee.get_protocol()

## Device operating mode

The `open()` process also reads the operating mode of the physical local device and stores it in the object. As with previous settings, you can retrieve the operating mode from the object at any time by calling the corresponding getter.

### Get the operating mode

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Get the operating mode of the device.
operating_mode = local_xbee.get_operating_mode()
```

Remote devices do not have an `open()` method, so you receive `UNKNOWN` when retrieving the operating mode of a remote XBee device.

The XBee Python Library supports two operating modes for local devices:

- API
- API with escaped characters

This means that AT (transparent) mode is not supported by the API. So, if you try to execute the `open()` method in a local device working in AT mode, you get an `XBeeException` caused by an `InvalidOperatingModeException`.

### 2.5.3.3 Close the XBee device connection

You must call the `close()` method each time you finish your XBee application. You can use this in the `finally` block or something similar.

If you don’t do this, you may have problems with the packet listener being executed in a separate thread.

This method guarantees that the listener thread will be stopped and the serial port will be closed.

### Close the connection

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)

try:
    xbee.open()

[...]

finally:
    if xbee is not None and xbee.is_open():
        xbee.close()
```
2.5.4 Configure the XBee device

One of the main features of the XBee Python Library is the ability to configure the parameters of local and remote XBee devices and execute some actions or commands on them.

To apply a complete configuration profile see Apply an XBee profile.

---

**Note:** Remote XBee devices cannot be opened, so they cannot be closed either. To close the connection of a remote device you need to close the connection of the local associated device.

---

**Warning:** The values set on the different parameters are not persistent through subsequent resets unless you store those changes in the device. For more information, see Write configuration changes.

### 2.5.4.1 Read and set common parameters

Local and remote XBee device objects provide a set of methods to get and set common parameters of the device. Some of these parameters are saved inside the XBee device object, and a cached value is returned when the parameter is requested. Other parameters are read directly from the physical XBee device when requested.

#### Cached parameters

Some parameters in an XBee device are used or requested frequently. To avoid the overhead of those parameters being read from the physical XBee device every time they are requested, they are saved inside the XBeeDevice object being returned when the getters are called.

The following table lists cached parameters and their corresponding getters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit address</td>
<td>get_64bit_addr()</td>
</tr>
<tr>
<td>16-bit address</td>
<td>get_16bit_addr()</td>
</tr>
<tr>
<td>Node identifier</td>
<td>get_node_id()</td>
</tr>
<tr>
<td>Firmware version</td>
<td>get_firmware_version()</td>
</tr>
<tr>
<td>Hardware version</td>
<td>get_hardware_version()</td>
</tr>
<tr>
<td>Role</td>
<td>get_role()</td>
</tr>
</tbody>
</table>

Local XBee devices read and save previous parameters automatically when opening the connection of the device. In remote XBee devices, you must issue the `read_device_info()` method to initialize the parameters.

You can refresh the value of those parameters (that is, read their values and update them inside the XBee device object) at any time by calling the `read_device_info()` method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_device_info(init=False)</td>
<td>Updates cache parameters reading them from the XBee: If init is True it reads all values, else only those not initialized.</td>
</tr>
</tbody>
</table>

**Refresh cached parameters**
### Instantiate an XBee device object.

```python
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()
```

### Refresh the cached values.

```python
local_xbee.refresh_device_info()
```

The `read_device_info()` method may fail for the following reasons:

- There is a timeout getting any of the device parameters, throwing a `TimeoutException`.
- The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
- The response of the command is not valid, throwing an `ATCommandException`.
- There is an error writing to the XBee interface, or device is closed, throwing a generic `XBeeException`.

All the cached parameters but the Node Identifier do not change; therefore, they cannot be set. For the Node Identifier, there is a method within all the XBee device classes that allows you to change it:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_node_id(String)</code></td>
<td>Specifies the new Node Identifier of the device. This method configures the physical XBee device with the provided Node Identifier and updates the cached value with the one provided.</td>
</tr>
</tbody>
</table>

### Non-cached parameters

The following non-cached parameters have their own methods to be configured within the XBee device classes:

- **Destination Address**: This setting specifies the default 64-bit destination address of a module that is used to report data generated by the XBee device (that is, IO sampling data). This setting can be read and set.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_dest_address()</code></td>
<td>Returns the 64-bit address of the device that data will be reported to.</td>
</tr>
<tr>
<td><code>set_dest_address(XBee64BitAddress)</code></td>
<td>Specifies the 64-bit address of the device where the data will be reported.</td>
</tr>
</tbody>
</table>

- **PAN ID**: This is the ID of the Personal Area Network the XBee device is operating in. This setting can be read and set.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_pan_id()</code></td>
<td>Returns a byte array containing the ID of the Personal Area Network where the XBee device is operating.</td>
</tr>
<tr>
<td><code>set_pan_id(Bytearray)</code></td>
<td>Specifies the value in byte array format of the PAN ID where the XBee device should work.</td>
</tr>
</tbody>
</table>

- **Power level**: This setting specifies the output power level of the XBee device. This setting can be read and set.
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_power_level()</code></td>
<td>Returns a <code>PowerLevel</code> enumeration entry indicating the power level of the XBee device.</td>
</tr>
<tr>
<td><code>set_power_level(PowerLevel)</code></td>
<td>Specifies a <code>PowerLevel</code> enumeration entry containing the desired output level of the XBee device.</td>
</tr>
</tbody>
</table>

#### Configure non-cached parameters

> # Instantiate an XBee device object.
> local_xbee = XBeeDevice("COM1", 9600)
> local_xbee.open()
>
> # Set the destination address of the device.
> dest_address = XBee64BitAddress.from_hex_string("0013A20040XXXXXX")
> local_xbee.set_dest_address(dest_address)
>
> # Read the operating PAN ID of the device.
> dest_addr = local_xbee.get_dst_address()
>
> # Read the operating PAN ID of the device.
> pan_id = local_xbee.get_pan_id()
>
> # Read the output power level.
> p_level = local_xbee.get_power_level()

All the previous getters and setters of the different options may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

#### Example: Common parameters

The XBee Python Library includes a sample application that displays how to get and set common parameters. It can be located in the following path:

`examples/configuration/ManageCommonParametersSample`

#### 2.5.4.2 Read, set and execute other parameters

If you want to read or set a parameter that does not have a custom getter or setter within the XBee device object, you can do so. All the XBee device classes (local or remote) include two methods to get and set any AT parameter, and a third one to run a command in the XBee device.
Get a parameter

You can read the value of any parameter of an XBee device using the `get_parameter()` method provided by all the XBee device classes. Use this method to get the value of a parameter that does not have its getter method within the XBee device object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_parameter(String)</code></td>
<td>Specifies the AT command (string format) to retrieve its value. The method returns the value of the parameter in a byte array.</td>
</tr>
</tbody>
</table>

Get a parameter from the XBee device

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Get the value of the Sleep Time (SP) parameter.
sp = local_xbee.get_parameter("SP")
```

The `get_parameter()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBBeeException`:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBBeeException`.

Example: Set and get parameters

The XBee Python Library includes a sample application that displays how to get and set parameters using the methods explained previously. It can be located in the following path:

`examples/configuration/SetAndGetParametersSample`

Set a parameter

To set a parameter that does not have its own setter method, you can use the `set_parameter()` method provided by all the XBee device classes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_parameter(String, Bytearray)</code></td>
<td>Specifies the AT command (String format) to be set in the device and a byte array containing the value of the parameter.</td>
</tr>
</tbody>
</table>

Set a parameter in the XBee device
# Instantiate an XBee device object.
llocal_xbee = XBeeDevice("COM1", 9600)
llocal_xbee.open()

# Configure the Node ID using the set_parameter() method.
llocal_xbee.set_parameter("NI", bytearray("Yoda", 'utf8'))

...}

The `set_parameter()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

### Example: Set and get parameters

The XBee Python Library includes a sample application that displays how to get and set parameters using the methods explained previously. It can be located in the following path:

`examples/configuration/SetAndGetParametersSample`

## Execute a command

There are other AT parameters that cannot be read or written. They are actions that are executed by the XBee device. The XBee Python library has several commands that handle most common executable parameters, but to run a parameter that does not have a custom command, you can use the `execute_command()` method provided by all the XBee device classes.

### Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execute_command(String)</code></td>
<td>Specifies the AT command (String format) to be run in the device.</td>
</tr>
</tbody>
</table>

## Run a command in the XBee device

...}

The `execute_command()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`: 
– The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
– The response of the command is not valid, throwing an ATCommandException.
– There is an error writing to the XBee interface, throwing a generic XBeeException.

2.5.4.3 Apply configuration changes

By default, when you perform any configuration on a local or remote XBee device, the changes are automatically applied. However, there could be some scenarios when you want to configure different settings or parameters of a device and apply the changes at the end when everything is configured. For that purpose, the XBeeDevice and RemoteXBeeDevice objects provide some methods that allow you to manage when to apply configuration changes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable_apply_changes()</td>
<td>Specifies whether the changes on settings and parameters are applied when set.</td>
<td>The apply configuration changes flag is enabled by default.</td>
</tr>
<tr>
<td>is_apply_changes_enabled()</td>
<td>Returns whether the XBee device is configured to apply parameter changes when they are set.</td>
<td></td>
</tr>
<tr>
<td>apply_changes()</td>
<td>Applies the changes on parameters that were already set but are pending to be applied.</td>
<td>This method is useful when the XBee device is configured to not apply changes when they are set.</td>
</tr>
</tbody>
</table>

Apply configuration changes

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Check if device is configured to apply changes.
apply_changes_enabled = local_xbee.is_apply_changes_enabled()

# Configure the device not to apply parameter changes automatically.
if apply_changes_enabled:
    local_xbee.enable_apply_changes(False)

# Set the PAN ID of the XBee device to BABE.
local_xbee.set_pan_id(utils.hex_string_to_bytes("BABE"))

# Perform other configurations.
[...]

# Apply changes.
local_xbee.apply_changes()
[...]
```

The apply_changes() method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.
- Other errors caught as XBeeException:
– The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
– The response of the command is not valid, throwing an ATCommandException.
– There is an error writing to the XBee interface, throwing a generic XBeeException.

### 2.5.4.4 Write configuration changes

If you want configuration changes performed in an XBee device to persist through subsequent resets, you need to write those changes in the device. Writing changes means that the parameter values configured in the device are written to the non-volatile memory of the XBee device. The module loads the parameter values from non-volatile memory every time it is started.

The XBee device classes (local and remote) provide a method to write (save) the parameter modifications in the XBee device memory so they persist through subsequent resets: write_changes().

```
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Set the PAN ID of the XBee device to BABE.
local_xbee.set_pan_id(utils.hex_string_to_bytes("BABE"))

# Perform other configurations.
[...]

# Apply changes.
local_xbee.apply_changes()

# Write changes.
local_xbee.write_changes()
[...]
```

The write_changes() method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.
- Other errors caught as XBeeException:
  - The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  - The response of the command is not valid, throwing an ATCommandException.
  - There is an error writing to the XBee interface, throwing a generic XBeeException.

### 2.5.4.5 Reset the device

It may be necessary to reset the XBee device when the system is not operating properly or you are initializing the system. All the XBee device classes of the XBee API provide the reset() method to perform a software reset on the local or remote XBee module.
In local modules, the `reset()` method blocks until a confirmation from the module is received, which usually takes one or two seconds. Remote modules do not send any kind of confirmation, so the method does not block when resetting them.

**Reset the module**

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Reset the module.
local_xbee.reset()
```

The `reset()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

**Example: Reset module**

The XBee Python Library includes a sample application that shows you how to perform a reset on your XBee device. The example is located in the following path:

`examples/configuration/ResetModuleSample`

---

### 2.5.4.6 Configure Wi-Fi settings

Unlike other protocols such as ZigBee or DigiMesh where devices are connected to each other, the XBee Wi-Fi protocol requires that the module is connected to an access point in order to communicate with other TCP/IP devices.

This configuration and connection with access points can be done using applications such as XCTU; however, the XBee Python Library includes a set of methods to configure the network settings, scan access points, and connect to an access point.

**Example: Configure Wi-Fi settings and connect to an access point**

The XBee Python Library includes a sample application that demonstrates how to configure the network settings of a Wi-Fi device and connect to an access point. You can locate the example in the following path:

`examples/configuration/ConnectToAccessPointSample`

---

**Configure IP addressing mode**

Before connecting your Wi-Fi module to an access point, you must decide how to configure the network settings using the IP addressing mode option. The supported IP addressing modes are contained in an enumerator called `IPAddressingMode`. It allows you to choose between:

- **DHCP**
• STATIC

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_ip_addressing_mode(IPAddressingMode)</td>
<td>Sets the IP addressing mode of the Wi-Fi module. Depending on the provided mode, network settings are configured differently:</td>
</tr>
<tr>
<td></td>
<td>• DHCP: Network settings are assigned by a server.</td>
</tr>
<tr>
<td></td>
<td>• STATIC: Network settings must be provided manually one by one.</td>
</tr>
</tbody>
</table>

Configure IP addressing mode

```python
[...]

# Instantiate an XBee device object.
local_xbee = WiFiDevice("COM1", 9600)
local_xbee.open()

# Configure the IP addressing mode to DHCP.
local_xbee.set_ip_addressing_mode(IPAddressingMode.DHCP)

# Save the IP addressing mode.
local_xbee.write_changes()
[...]```

The `set_ip_addressing_mode()` method may fail for the following reasons:

• There is a timeout setting the IP addressing parameter, throwing a `TimeoutException`.

• Other errors caught as `XBeeException`:
  
    − The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
    
    − The response of the command is not valid, throwing an `ATCommandException`.
    
    − There is an error writing to the XBee interface, throwing a generic `XBeeException`.

Configure IP network settings

Like any TCP/IP protocol device, the XBee Wi-Fi modules have the IP address, subnet mask, default gateway and DNS settings that you can get at any time using the XBee Python Library.

Unlike some general configuration settings, these parameters are not saved inside the WiFiDevice object. Every time you request the parameters, they are read directly from the Wi-Fi module connected to the computer. The following parameters are used in the configuration of the TCP/IP protocol:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP address</td>
<td><code>get_ip_address()</code></td>
</tr>
<tr>
<td>Subnet mask</td>
<td><code>get_mask_address()</code></td>
</tr>
<tr>
<td>Gateway IP</td>
<td><code>get_gateway_address()</code></td>
</tr>
<tr>
<td>DNS address</td>
<td><code>get_dns_address()</code></td>
</tr>
</tbody>
</table>

Read IP network settings
You can also change those settings when the module has static IP configuration with the following methods:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP address</td>
<td>set_ip_addr()</td>
</tr>
<tr>
<td>Subnet mask</td>
<td>set_mask_address()</td>
</tr>
<tr>
<td>Gateway IP</td>
<td>set_gateway_address()</td>
</tr>
<tr>
<td>DNS address</td>
<td>set_dns_address()</td>
</tr>
</tbody>
</table>

### 2.5.4.7 Configure Bluetooth settings

Newer XBee3 devices have a Bluetooth® Low Energy (BLE) interface that enables you to connect your XBee device to another device such as a cellphone. The XBee device classes (local and remote) offer some methods that allow you to:

- *Enable and disable Bluetooth*
- *Configure the Bluetooth password*
- *Read the Bluetooth MAC address*

**Enable and disable Bluetooth**

Before connecting to your XBee device over Bluetooth Low Energy, you first have to enable this interface. The XBee Python Library provides a couple of methods to enable or disable this interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable_bluetooth()</td>
<td>Enables the Bluetooth Low Energy interface of your XBee device.</td>
</tr>
<tr>
<td>disable_bluetooth()</td>
<td>Disables the Bluetooth Low Energy interface of your XBee device.</td>
</tr>
</tbody>
</table>

**Enabling and disabling the Bluetooth interface**

([...])
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Enable the Bluetooth interface.
local_xbee.enable_bluetooth()

[...]

# Disable the Bluetooth interface.
local_xbee.disable_bluetooth()

[...]

These methods may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.
- Other errors caught as XBeeException:
  - The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  - The response of the command is not valid, throwing an ATCommandException.
  - There is an error writing to the Xbee interface, throwing a generic XBeeException.

## Configure the Bluetooth password

Once you have enabled the Bluetooth Low Energy, you must configure the password you will use to connect to the device over that interface (if not previously done). For this purpose, the API offers the following method:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>update_bluetooth_password(String)</td>
<td>Specifies the new Bluetooth password of the XBe device.</td>
</tr>
</tbody>
</table>

### Configuring or changing the Bluetooth password

[...]  

# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

new_password = "myBluetoothPassword" # Do not hard-code it in the app!

# Configure the Bluetooth password.
local_xbee.update_bluetooth_password(new_password)

[...]

The update_bluetooth_password method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.
- Other errors caught as XBeeException:
  - The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
– The response of the command is not valid, throwing an ATCommandException.
– There is an error writing to the XBee interface, throwing a generic XBeeException.

**Warning:** Never hard-code the Bluetooth password in the code, a malicious person could decompile the application and find it out.

### Read the Bluetooth MAC address

Another method that the XBee Java Library provides is `get_bluetooth_mac_addr()`, which returns the EUI-48 Bluetooth MAC address of your XBee device in a format such as “00112233AABB”.

#### Reading the Bluetooth MAC address

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

print("The Bluetooth MAC address is: \$s" % local_xbee.get_bluetooth_mac_addr())
```

The `get_bluetooth_mac_addr` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.
- Other errors caught as XBeeException:
  - The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  - The response of the command is not valid, throwing an ATCommandException.
  - There is an error writing to the XBee interface, throwing a generic XBeeException.

#### 2.5.5 Discover the XBee network

Several XBee modules working together and communicating with each other form a network. XBee networks have different topologies and behaviors depending on the protocol of the XBee devices that form it.

The XBee Python Library includes a class, called XBeeNetwork, that represents the set of nodes forming the actual XBee network. This class allows you to perform some operations related to the nodes.

**Note:** There are XBeeNetwork subclasses for different protocols which correspond to the XBeeDevice subclasses:

- XBee ZigBee network (ZigBeeNetwork)
- XBee 802.15.4 network (Raw802Network)
- XBee DigiMesh network (DigiMeshNetwork)
- XBee DigiPoint network (DigiPointNetwork)
The XBee Network object can be retrieved from a local XBee device after it has been opened using the `get_network()` method.

**Warning:** Because XBee Cellular and Wi-Fi module protocols are directly connected to the Internet and do not share a connection, these protocols do not support XBee networks.

## Retrieve the XBee network

```python
# Instantiate an XBee device object.
xbee = XBeeDevice("COM1", 9600)
xbee.open()

# Get the network.
xnet = xbee.get_network()
```

A main feature of the `XBeeNetwork` class is the ability to discover the XBee devices that form the network. The `XBeeNetwork` object provides the following operations related to the XBee devices discovery feature:

- Configure the discovery process
- Discover the network
- Access the discovered devices
- Add and remove devices manually
- Listen to network modification events

### 2.5.5.1 Configure the discovery process

Before discovering all the nodes of a network, you can configure the settings of that process. The API provides two methods to configure the discovery timeout and discovery options. These methods set the values in the module.
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_discovery_timeout(Float)</code></td>
<td>Configures the discovery timeout (NT parameter) with the given value in seconds.</td>
</tr>
</tbody>
</table>
| `set_discovery_options(Set<DiscoveryOptions>)` | Configures the discovery options (NO parameter) with the set of options. The set of discovery options contains the different `DiscoveryOptions` configuration values that are applied to the local XBee module when performing the discovery process. These options are the following:

  - **DiscoveryOptions.APPEND_DD**: Appends the device type identifier (DD) to the information retrieved when a node is discovered. This option is valid for DigiMesh, Point-to-multipoint (Digi Point) and ZigBee protocols.
  
  - **DiscoveryOptions.DISCOVER_MYSELF**: The local XBee device is returned as a discovered device. This option is valid for all protocols.
  
  - **DiscoveryOptions.APPEND_RSSI**: Appends the RSSI value of the last hop to the information retrieved when a node is discovered. This option is valid for DigiMesh and Point-to-multipoint (Digi Point) protocols. |

#### Configure discovery timeout and options

```python
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

# Get the network.
xnet = xbee.get_network()  

# Configure the discovery options.
xnet.set_discovery_options({DiscoveryOptions.DISCOVER_MYSELF, DiscoveryOptions.APPEND_DD})

# Configure the discovery timeout, in SECONDS.
xnet.set_discovery_timeout(25)
```

#### 2.5.5.2 Discover the network

The `XBeeNetwork` object discovery process allows you to discover and store all the XBee devices that form the network. The `XBeeNetwork` object provides a method for executing the discovery process:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start_discovery_process</code></td>
<td>Starts the discovery process, saving the remote XBee devices found inside the <code>XBeeNetwork</code> object.</td>
</tr>
</tbody>
</table>
When a discovery process has started, you can monitor and manage it using the following methods provided by the XBeeNetwork object:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_discovery_running()</td>
<td>Returns whether or not the discovery process is running.</td>
</tr>
<tr>
<td>stop_discovery_process()</td>
<td>Stops the discovery process that is taking place.</td>
</tr>
</tbody>
</table>

**Warning:** Although you call the `stop_discovery_process` method, DigiMesh and DigiPoint devices are blocked until the configured discovery time has elapsed. If you try to get or set any parameter during that time, a `TimeoutException` is thrown.

Once the process has finished, you can retrieve the list of devices that form the network using the `get_devices()` method provided by the network object. If the discovery process is running, this method returns `None`.

All discovered XBee devices are stored in the `XBeeNetwork` instance.

**Discover the network**

```python
[...]
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

# Get the XBee Network object from the XBee device.
xnet = xbee.get_network()  

# Start the discovery process and wait for it to be over.
xnet.start_discovery_process()  
while xnet.is_discovery_running():  
    time.sleep(0.5)  

# Get a list of the devices added to the network.
devices = xnet.get_devices()  
[...]
```

**Discover the network with an event notification**

The API also allows you to add a discovery event listener to notify you when new devices are discovered, the process finishes, or an error occurs during the process. In this case, you must provide an event listener before starting the discovery process using the `add_device_discovered_callback()` method.

**Add a callback to device discovered event**

```python
[...]
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

# Define the device discovered callback.
def callback(remote):  
    [...]
```

(continues on next page)
# Get the XBee Network object from the XBee device.
xnet = xbee.get_network()

# Add the device discovered callback.
xnet.add_device_discovered_callback(callback)

# Start the discovery process.
xnet.start_discovery_process()

The behavior of the event is as follows:

- When a new remote XBee device is discovered, the `DeviceDiscovered` event is raised, executing all device discovered callbacks, even if the discovered device is already in the devices list of the network. The callback receives a `RemoteXBeeDevice` as argument, with all available information. Unknown parameters of this remote device will be `None`.

There is also another event, `DiscoveryProcessFinished`. This event is raised all times that a discovery process finishes.

## Add a callback to discovery process finished event

```python
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

def callback(status):
    if status == NetworkDiscoveryStatus.ERROR_READ_TIMEOUT:
        [...]

# Add the discovery process finished callback.
xnet.add_discovery_process_finished_callback(callback)

The behavior of the event is as follows:

- When a discovery process has finished for any reason (either successfully or with an error), this event is raised, and all callbacks associated with it are executed. This method receives a `NetworkDiscoveryStatus` object as parameter. This status represents the result of the network discovery process.

**Example: Device discovery**

The XBee Python Library includes a sample application that displays how to perform a device discovery using a callback. It can be located in the following path:

`examples/network/DiscoverDevicesSample/DiscoverDevicesSample.py`

**Discover specific devices**

The `XBeeNetwork` object also provides methods to discover specific devices within a network. This is useful, for example, if you only need to work with a particular remote device.
Discover specific devices

```
...]
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  
...]
# Get the XBee Network object from the XBee device.
xnet = xbee.get_network()  
# Discover the remote device whose node ID is 'SOME NODE ID'.
remote = xnet.discover_device("SOME NODE ID")  
# Discover the remote devices whose node IDs are 'ID 2' and 'ID 3'.
remote_list = xnet.discover_devices(['ID 2', 'ID 3'])
...]
```

### 2.5.5.3 Access the discovered devices

Once a discovery process has finished, the discovered nodes are saved inside the XBeeNetwork object. This means that you can get a list of discovered devices at any time. Using the `get_devices()` method you can obtain all the devices in this list, as well as work with the list object as you would with other lists.

This is the list of methods provided by the XBeeNetwork object that allow you to retrieve already discovered devices:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_devices(String)</code></td>
<td>Returns a copy of the list of remote XBee devices. If some device is added to the network before calling this method, the list returned will not be updated.</td>
</tr>
<tr>
<td><code>get_device_by_64(XBee64BitAddress)</code></td>
<td>Remote device already contained in the network whose 64-bit address matches the given one or None if the device is not in the network.</td>
</tr>
<tr>
<td><code>get_device_by_16(XBee16BitAddress)</code></td>
<td>Remote device already contained in the network whose 16-bit address matches the given one or None if the device is not in the network.</td>
</tr>
<tr>
<td><code>get_device_by_node_id(String)</code></td>
<td>The remote device already contained in the network whose node identifier matches the given one or None if the device is not in the network.</td>
</tr>
</tbody>
</table>

Access discovered devices

---

Note: These methods are blocking, so the application will block until the devices are found or the configured timeout expires.
2.5.4 Add and remove devices manually

This section provides information on methods for adding, removing, and clearing the list of remote XBee devices.

Manually add devices to the XBee network

There are several methods for adding remote XBee devices to an XBee network, in addition to the discovery methods provided by the `XBeeNetwork` object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_remote(RemoteXBeeDevice)</code></td>
<td>device to be added to the list of remote devices of the <code>XBeeNetwork</code> object.</td>
</tr>
<tr>
<td>Notice</td>
<td>that this operation does not join the remote XBee device to the network; it just tells the network that it contains that device. However, the device has only been added to the device list, and may not be physically in the same network.</td>
</tr>
<tr>
<td>Note</td>
<td>that if the given device already exists in the network, it won’t be added, but the device in the current network will be updated with the not None parameters of the given device. This method returns the given device with the parameters updated. If the device was not in the list yet, this method returns it without changes.</td>
</tr>
</tbody>
</table>

Add a remote device manually to the network

[...]

(continues on next page)
Remove an existing device from the XBee network

It is also possible to remove a remote XBee device from the list of remote XBee devices of the XBeeNetwork object by calling the following method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove_device</td>
<td>Specifies the remote XBee device to be removed from the list of remote devices of the XBeeNetwork object. If the device is not contained in the list, the method will raise a ValueError. Notice that this operation does not remove the remote XBee device from the actual XBee network; it just tells the network object that it will no longer contain that device. However, next time you perform a discovery, it could be added again automatically.</td>
</tr>
</tbody>
</table>

Remove a remote device from the network

```python
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  
[...

# Get the XBee Network object from the XBee device.
xnet = xbee.get_network()  

# Get the remote XBee device.
remote = xnet.get_remote(...) 

# Add the remote device to the network.
xnet.add_remote(remote) 

[...]
```
Clear the list of remote XBee devices from the XBee network

The XBeeNetwork object also includes a method to clear the list of remote devices. This can be useful when you want to perform a clean discovery, cleaning the list before calling the discovery method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear()</td>
<td>Removes all the devices from the list of remote devices of the network. Notice that this does not imply removing the XBee devices from the actual XBee network; it just tells the object that the list should be empty now. Next time you perform a discovery, the list could be filled with the remote XBee devices found.</td>
</tr>
</tbody>
</table>

Clear the list of remote devices

```python
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

# Get the XBee Network object from the XBee device.
xnet = xbee.get_network()  

# Discover devices in the network and add them to the list of devices.  

# Clear the list of devices.
xnet.clear()  
```

2.5.5.5 Listen to network modification events

When a discovery process finds new nodes that were not in the XBee network cache (XBeeNetwork or a subclass), they are stored generating a modification in the XBee network object. A manual removal or addition of an XBee to the network also causes a modification.

The XBee library notifies about these network cache modification events to registered callbacks. These events inform about network modifications:

- Addition of new nodes
- Removal of existing nodes
- Update of nodes
- Network clear

To receive any of these modification events you must provide a callback using the add_network_modified_callback() method. This callback must follow the format:

```python
def my_callback(event_type, reason, node):
    
    """
    Callback to notify about a new network modification event.
    """

    Args:
```
When a modification in the network cache occurs, all network modification callbacks are executed. Each callback receives the following arguments:

- The type of network modification as a `NetworkEventType` (addition, removal, update or clear)
- The modification cause as a `NetworkEventReason` (discovered, received message, manual)
- The XBee node, local or remote, (`AbstractXBeeDevice`) involved in the modification (None for a clear event type)

Register a network modifications callback

```python
# Define the network modified callback.
def cb_network_modified(event_type, reason, node):
    print(" >>>> Network event:")
    print(" Type: \%s (\%d)" % (event_type.description, event_type.code))
    print(" Reason: \%s (\%d)" % (reason.description, reason.code))
    if not node:
        return
    print(" Node:")
    print(" \%s" % node)

xnet = xbee.get_network()
# Add the network modified callback.
xnet.add_network_modified_callback(cb_network_modified)
```

**Network events**

The `NetworkEventType` class enumerates the possible network cache modification types:

- **Addition** (`NetworkEventType.ADD`): A new XBee has just been added to the network cache.
- **Deletion** (`NetworkEventType.DEL`): An XBee in the network cache has just been removed.
- **Update** (`NetworkEventType.UPDATE`): An existing XBee in the network cache has just been updated. This means any of its parameters (node id, 16-bit address, role, ...) changed.
- **Clear** (`NetworkEventType.CLEAR`): The network cached has just been cleared.

As well, `NetworkEventReason` enumerates the network modification causes:

- **NetworkEventReason.DISCOVERED**: The device was added/removed/updated during a discovery process.
- **NetworkEventReason.RECEIVED_MSG**: The device was added after receiving a message from it.
NetworkEventReason.MANUAL: The device was manually added/removed.

For example, if, during a discovery process, a new device is found and:

- it is not in the network cache yet, the addition triggers a new event with:
  - type: NetworkEventType.ADD
  - cause: NetworkEventReason.DISCOVERED
- it is already in the network cache but its node identifier is updated, a new event is raised with:
  - type: NetworkEventType.UPDATE
  - cause: NetworkEventReason.DISCOVERED
- it is already in the network and nothing has changed, no event is triggered.

Example: Network modifications

The XBee Python Library includes a sample application that displays how to receive network modification events. It can be located in the following path:

```
extamples/network/NetworkModificationsSample/NetworkModificationsSample.py
```

2.5.6 Communicate with XBee devices

The XBee Python Library provides the ability to communicate with remote nodes in the network, IoT devices and other interfaces of the local device. The communication between XBee devices in a network involves the transmission and reception of data.

Warning: Communication features described in this topic and sub-topics are only applicable for local XBee devices. Remote XBee device classes do not include methods for transmitting or receiving data.

2.5.6.1 Send and receive data

XBee modules can communicate with other devices that are on the same network and use the same radio frequency. The XBee Python Library provides several methods to send and receive data between the local XBee device and any remote on the network.

- **Send data**
- **Receive data**

Send data

A data transmission operation sends data from your local (attached) XBee device to a remote device on the network. The operation sends data in API frames, but the XBee Python library abstracts the process so you only need to specify the device you want to send data to and the data itself.

You can send data either using a unicast or broadcast transmission. Unicast transmissions route data from one source device to one destination device, whereas broadcast transmissions are sent to all devices in the network.
**Send data to one device**

Unicast transmissions are sent from one source device to another destination device. The destination device could be an immediate neighbor of the source, or it could be several hops away.

Data transmission can be synchronous or asynchronous, depending on the method used.

**Synchronous operation**

This type of operation is blocking. This means the method waits until the transmit status response is received or the default timeout is reached.

The `XBeeDevice` class of the API provides the following method to perform a synchronous unicast transmission with a remote node of the network:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send_data(RemoteXBeeDevice, String or Bytearray, Integer)</code></td>
<td>Specifies the remote XBee destination object, the data to send and optionally the transmit options.</td>
</tr>
</tbody>
</table>

Protocol-specific classes offer additional synchronous unicast transmission methods apart from the one provided by the `XBeeDevice` object:

<table>
<thead>
<tr>
<th>XBee class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZigBeeDevice</td>
<td><code>send_data_64_16(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)</code></td>
<td>Specifies the 64-bit and 16-bit destination addresses, the data to send and optionally the transmit options. If you do not know the 16-bit address, use the <code>XBee16BitAddress.UNKNOWN_ADDRESS</code>.</td>
</tr>
<tr>
<td>Raw802Device</td>
<td><code>send_data_16(XBee16BitAddress, String or Bytearray, Integer)</code></td>
<td>Specifies the 16-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td></td>
<td><code>send_data_64(XBee64BitAddress, String or Bytearray, Integer)</code></td>
<td>Specifies the 64-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td>DigiMeshDevice</td>
<td><code>send_data_64(XBee64BitAddress, String or Bytearray, Integer)</code></td>
<td>Specifies the 64-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td>DigiPointDevice</td>
<td><code>send_data_64_16(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)</code></td>
<td>Specifies the 64-bit and 16-bit destination addresses, the data to send and optionally the transmit options. If you do not know the 16-bit address, use the <code>XBee16BitAddress.UNKNOWN_ADDRESS</code>.</td>
</tr>
</tbody>
</table>

**Send data synchronously**

```python
[...,]

# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()

# Instantiate a remote XBee device object.
remote_device = RemoteXBeeDevice(device, XBee64BitAddress.from_hex_string("0013A20040XXXXXX"))

# Send data using the remote object.
device.send_data(remote_device, "Hello XBee!")

[...]
```
The previous methods may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a \texttt{TimeoutException}.
- Other errors caught as \texttt{XBeeException}:
  - The operating mode of the device is not API or ESCAPED\_API\_MODE, throwing an \texttt{InvalidOperatingModeException}.
  - The response of the command is not valid, throwing an \texttt{ATCommandException}.
  - There is an error writing to the XBee interface, throwing a generic \texttt{XBeeException}.

The default timeout to wait for the send status is two seconds. However, you can configure the timeout using the \texttt{get\_sync\_ops\_timeout} and \texttt{set\_sync\_ops\_timeout} methods of an XBee device class.

\textbf{Get/set the timeout for synchronous operations}

\begin{verbatim}
[#]
NEW_TIMEOUT\_FOR\_SYNC\_OPERATIONS = 5 \# 5 seconds

device = [...] # Retrieving the configured timeout for synchronous operations.
print("Current timeout: \%d seconds\" % device.get_sync_ops_timeout())
[#]
# Configuring the new timeout (in seconds) for synchronous operations.
device.set_sync_ops_timeout(NEW_TIMEOUT\_FOR\_SYNC\_OPERATIONS)
[#]
\end{verbatim}

**Example: Synchronous unicast transmission**

The XBee Python Library includes a sample application that shows you how to send data to another XBee device on the network. The example is located in the following path:

\texttt{examples/communication/SendDataSample}

**Asynchronous operation**

Transmitting data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was successfully sent to the remote device.

The \texttt{XBeeDevice} class of the API provides the following method to perform an asynchronous unicast transmission with a remote node on the network:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{send_data_async(RemoteXBeeDevice, String or Bytestring, Integer)}</td>
<td>Specifies the remote XBee destination object, the data to send and optionally the transmit options.</td>
</tr>
</tbody>
</table>

Protocol-specific classes offer some other asynchronous unicast transmission methods in addition to the one provided by the \texttt{XBeeDevice} object:
### XBee class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send_data_async_64_16()</code></td>
<td>Specifies the 64-bit and 16-bit destination addresses, the data to send and optionally the transmit options. If you do not know the 16-bit address, use <code>XBee16BitAddress.UNKNOWN_ADDRESS</code>.</td>
</tr>
<tr>
<td><code>send_data_async_16()</code></td>
<td>Specifies the 16-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td><code>send_data_async_64()</code></td>
<td>Specifies the 64-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td><code>send_data_async_64_16()</code></td>
<td>Specifies the 64-bit and 16-bit destination addresses, the data to send and optionally the transmit options. If you do not know the 16-bit address, use <code>XBee16BitAddress.UNKNOWN_ADDRESS</code>.</td>
</tr>
</tbody>
</table>

### Send data asynchronously

```python
[...] # Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()

# Instantiate a remote XBee device object.
remote_device = RemoteXBeeDevice(device, XBee64BitAddress.from_hex_string("0013A20040XXXXXX"))

device.send_data_async(remote_device, "Hello XBee!")
[...]```

The previous methods may fail for the following reasons:

- All the possible errors are caught as an `XBeeException`:
  - The operating mode of the device is not API or ESCAPED_API_MODE, throwing an `InvalidOperatingModeException`.
  - There is an error writing to the XBe device interface, throwing a generic `XBeeException`.

### Example: Asynchronous unicast transmission

The XBee Python Library includes a sample application that shows you how to send data to another XBee device asynchronously. The example is located in the following path:

`examples/communication/SendDataAsyncSample`

### Send data to all devices of the network

Broadcast transmissions are sent from one source device to all the other devices on the network.

All the XBee device classes (generic and protocol specific) provide the same method to send broadcast data:
## Send broadcast data

The `send_data_broadcast` method may fail for the following reasons:

- Transmit status is not received in the configured timeout, throwing a `TimeoutException` exception.
- Error types catch as `XBeeException`:
  - The operating mode of the device is not `API` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The transmit status is not `SUCCESS`, throwing a `TransmitException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

### Example: Broadcast transmission

The XBee Python Library includes a sample application that shows you how to send data to all the devices on the network (broadcast). The example is located in the following path:

```
exampless/communication/SendBroadcastDataSample
```

## Receive data

The data reception operation allows you to receive and handle data sent by other remote nodes of the network.

There are two different ways to read data from the device:

- **Polling for data.** This mechanism allows you to read (ask) for new data in a polling sequence. The read method blocks until data is received or until a configurable timeout has expired.
- **Data reception callback.** In this case, you must register a listener that executes a callback each time new data is received by the local XBee device (that is, the device attached to your PC) providing data and other related information.

### Polling for data

The simplest way to read for data is by executing the `read_data` method of the local XBee device. This method blocks your application until data from any XBee device of the network is received or the timeout provided has expired:

```python
# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()

# Send broadcast data.
device.send_data_broadcast("Hello XBees!")
```

The `send_data_broadcast` method may fail for the following reasons:
Method | Description
---|---
read_data(Integer) | the time to wait for data reception (method blocks during that time and throws a TimeoutException if no data is received). If you do not specify a timeout, the method returns immediately the read message or None if the device did not receive new data.

Reading data from any remote XBee device (polling)

```python
# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()

# Read data.
xbee_message = device.read_data()
```

The method returns the read data inside an XBeeMessage object. This object contains the following information:

- RemoteXBeeDevice that sent the message.
- Byte array with the contents of the received data.
- Flag indicating if the data was sent via broadcast.
- Time when the message was received.

You can retrieve the previous information using the corresponding attributes of the XBeeMessage object:

Get the XBeeMessage information

```python
xbree_message = device.read_data()
remote_device = xbee_message.remote_device
data = xbee_message.data
is_broadcast = xbee_message.is_broadcast
timestamp = xbee_message.timestamp
```

You can also read data from a specific remote XBee device of the network. For that purpose, the XBee device object provides the read_data_from method:

Method | Description
---|---
read_data_from(RemoteXBeeDevice, Integer) | XBee device to read data from and the time to wait for data reception (method blocks during that time and throws a TimeoutException if no data is received). If you do not specify a timeout, the method returns immediately the read message or None if the device did not receive new data.

Read data from a specific remote XBee device (polling)

```python
# Instantiate an XBee device object.
```
device = XBeeDevice("COM1", 9600)
device.open()

# Instantiate a remote XBee device object.
remote_device = RemoteXBeeDevice(device, XBee64BitAddress.from_hex_string("0013A200XXXXXX"))

# Read data sent by the remote device.
xbee_message = device.read_data(remote_device)

[...]
To stop listening to new received data, use the `del_data_received_callback` method to unsubscribe the already-registered callback.

**Deregister data reception**

```python
def my_data_received_callback(xbee_message):
    [...]  
device.add_data_received_callback(my_data_received_callback)
    [...]  
# Delete the callback
device.del_data_received_callback(my_data_received_callback)
    [...]  
```

**Example: Register for data reception**

The XBee Python Library includes a sample application that shows you how to subscribe to the data reception service to receive data. The example is located in the following path:

`examples/communication/ReceiveDataSample`

### 2.5.6.2 Send and receive explicit data

Some ZigBee applications may require communication with third-party (non-Digi) RF modules. These applications often send and receive data of different public profiles such as Home Automation or Smart Energy to other modules.

Xbee ZigBee modules offer a special type of frame for this purpose. Explicit frames are used to transmit and receive explicit data. When sending public profile packets, the frames transmit the data itself plus the application layer-specific fields—the source and destination endpoints, profile ID, and cluster ID.

**Warning:** Only ZigBee, DigiMesh, and Point-to-Multipoint protocols support the transmission and reception of data in explicit format. This means you cannot transmit or receive explicit data using a generic `XBeeDevice` object. You must use a protocol-specific XBe device object such as a `ZigBeeDevice`.

- Send explicit data
- Receive explicit data

### Send explicit data

You can send explicit data as either unicast or broadcast transmissions. Unicast transmissions route data from one source device to one destination device, whereas broadcast transmissions are sent to all devices in the network.

#### Send explicit data to one device

Unicast transmissions are sent from one source device to another destination device. The destination device could be an immediate neighbor of the source, or it could be several hops away.

Unicast explicit data transmission can be a synchronous or asynchronous operation, depending on the method used.
Synchronous operation

The synchronous data transmission is a blocking operation. That is, the method waits until it either receives the transmit status response or the default timeout is reached.

All local XBees device classes that support explicit data transmission provide a method to transmit unicast and synchronous explicit data to a remote node of the network:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_expl_data(RemoteXBeeDevice, Integer, Integer, Integer, Integer, String or bytearray, Integer)</td>
<td>Specifies remote XBees destination object, four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options.</td>
</tr>
</tbody>
</table>

Send unicast explicit data synchronously

```python
[# Instantiate a ZigBee device object.
device = ZigBeeDevice("COM1", 9600)
device.open()

# Instantiate a remote ZigBee device object.
remote_device = RemoteZigBeeDevice(device, XBee64BitAddress.from_hex_string("0013A20040XXXXXX"))

# Send explicit data using the remote object.
device.send_expl_data(remote_device, 0xA0, 0xA1, 0x1554, 0xC105, "Hello XBee!")
```

The previous methods may fail for the following reasons:

- The method throws a `TimeoutException` exception if the response is not received in the configured timeout.
- Other errors register as `XBeeException`:
  - If the operating mode of the device is not API or `ESCAPED_API_MODE`, the method throws an `InvalidOperatingModeException`.
  - If the transmit status is not `SUCCESS`, the method throws a `TransmitException`.
  - If there is an error writing to the XBee interface, the method throws a generic `XBeeException`.

The default timeout to wait for the send status is two seconds. However, you can configure the timeout using the `get_sync_ops_timeout` and `set_sync_ops_timeout` methods of an XBees device class.

Example: Transmit explicit synchronous unicast data

The XBees Python Library includes a sample application that demonstrates how to send explicit data to a remote device of the network (unicast). It can be located in the following path:

```bash
examples/communication/explicit/SendExplicitDataSample
```

Asynchronous operation

Transmitting explicit data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was successfully sent to the remote device.
All local XBee device classes that support explicit data transmission provide a method to transmit unicast and asynchronous explicit data to a remote node of the network:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send_expl_data_async(RemoteXBeeDevice, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</code></td>
<td>Specifies remote XBee destination object, four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options.</td>
</tr>
</tbody>
</table>

The previous methods may fail for the following reasons:

- All the possible errors are caught as an `XBeeException`:
  - The operating mode of the device is not `API` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

**Example: Transmit explicit asynchronous unicast data**

The XBee Python Library includes a sample application that demonstrates how to send explicit data to other XBee devices asynchronously. It can be located in the following path:

`examples/communication/explicit/SendExplicitDataAsyncSample`

**Send explicit data to all devices in the network**

Broadcast transmissions are sent from one source device to all other devices in the network.

All protocol-specific XBee device classes that support the transmission of explicit data provide the same method to send broadcast explicit data:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send_expl_data_broadcast(Integer, Integer, Integer, Integer, String or Bytearray, Integer)</code></td>
<td>Specifies the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options.</td>
</tr>
</tbody>
</table>

Send broadcast data
# Instantiate a ZigBee device object.
device = ZigBeeDevice("COM1", 9600)
device.open()

# Send broadcast data.
device.send_expl_data_broadcast(0xA0, 0xA1, 0x1554, 0xC105, "Hello XBees!")

The send_expl_data_broadcast method may fail for the following reasons:

- Transmit status is not received in the configured timeout, throwing a TimeoutException exception.
- Error types catch as XBeeException:
  - The operating mode of the device is not API or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  - The transmit status is not SUCCESS, throwing a TransmitException.
  - There is an error writing to the XBee interface, throwing a generic XBeeException.

Example: Send explicit broadcast data

The XBee Python Library includes a sample application that demonstrates how to send explicit data to all devices in the network (broadcast). It can be located in the following path:
examples/communication/explicit/SendBroadcastExplicitDataSample

Receive explicit data

Some applications developed with the XBee Python Library may require modules to receive data in application layer, or explicit, data format.

To receive data in explicit format, you must first configure the data output mode of the receiver XBee device to explicit format using the set_api_output_mode_value method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_api_output_mode_value()</td>
<td>Returns the API output mode of the data received by the XBee device.</td>
</tr>
<tr>
<td>set_api_output_mode_value(value)</td>
<td>Specifies the API output mode of the data received by the XBee device. Calculate the mode with the method calculate_api_output_mode_value with a set of APIOutputModeBit.</td>
</tr>
</tbody>
</table>

Set API output mode

[...]

# Instantiate a ZigBee device object.
device = ZigBeeDevice("COM1", 9600)
device.open()

# Set explicit output mode
mode = APIOutputModeBit.calculate_api_output_mode_value(device.get_protocol(),
    {APIOutputModeBit.EXPLICIT})
device.set_api_output_mode_value(mode)
Once you have configured the device to receive data in explicit format, you can read it using one of the following mechanisms provided by the XBee device object.

### Polling for explicit data

The simplest way to read for explicit data is by executing the `read_expl_data` method of the local XBee device. This method blocks your application until explicit data from any XBee device of the network is received or the provided timeout has expired:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_expl_data(Integer)</code></td>
<td>Specifies the time to wait in seconds for explicit data reception (method blocks during that time and throws a <code>TimeoutException</code> if no data is received). If you do not specify a timeout, the method returns immediately the read message or <code>None</code> if the device did not receive new data.</td>
</tr>
</tbody>
</table>

#### Read explicit data from any remote XBee device (polling)

```python
# Instantiate a ZigBee device object.
device = ZigBeeDevice("COM1", 9600)
device.open()

# Read data.
xbee_message = device.read_expl_data()
```

The method returns the read data inside an `ExplicitXBeeMessage` object. This object contains the following information:

- `RemoteXBeeDevice` that sent the message.
- Endpoint of the source that initiated the transmission.
- Endpoint of the destination where the message is addressed.
- Cluster ID where the data was addressed.
- Profile ID where the data was addressed.
- Byte array with the contents of the received data.
- Flag indicating if the data was sent via broadcast.
- Time when the message was received.
You can retrieve the previous information using the corresponding attributes of the `ExplicitXBeeMessage` object:

**Get the ExplicitXBeeMessage information**

```python
expl_xbee_message = device.read_expl_data()
remote_device = expl_xbee_message.remote_device
source_endpoint = expl_xbee_message.source_endpoint
dest_endpoint = expl_xbee_message.dest_endpoint
cluster_id = expl_xbee_message.cluster_id
profile_id = expl_xbee_message.profile_id
data = xbee_message.data
is_broadcast = expl_xbee_message.is_broadcast
timestamp = expl_xbee_message.timestamp
```

You can also read explicit data from a specific remote XBee device of the network. For that purpose, the XBee device object provides the `read_expl_data_from` method:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_expl_data_from(RemoteXBeeDevice, Integer)</code></td>
<td>Device to read explicit data from and the time to wait for explicit data reception (method blocks during that time and throws a <code>TimeoutException</code> if no data is received). If you do not specify a timeout, the method returns immediately the read message or <code>None</code> if the device did not receive new data.</td>
</tr>
</tbody>
</table>

**Read explicit data from a specific remote XBee device (polling)**

```python
# Instantiate a ZigBee device object.
device = ZigBeeDevice("COM1", 9600)
device.open()

# Instantiate a remote ZigBee device object.
remote_device = RemoteZigBeeDevice(device, XBee64BitAddress.from_hex_string("0013A200XXXXXX"))

# Read data sent by the remote device.
expl_xbee_message = device.read_expl_data(remote_device)
```

As in the previous method, this method also returns an `ExplicitXBeeMessage` object with all the information inside.

The default timeout to wait for data is two seconds. However, you can configure the timeout using the `get_sync_ops_timeout` and `set_sync_ops_timeout` methods of an XBee device class.

**Example: Receive explicit data with polling**

The XBee Python Library includes a sample application that demonstrates how to receive explicit data using the polling mechanism. It can be located in the following path:

`examples/communication/explicit/ReceiveExplicitDataPollingSample`
Explicit data reception callback

This mechanism for reading explicit data does not block your application. Instead, you can be notified when new explicit data has been received if you are subscribed or registered to the explicit data reception service by using the `add_expl_data_received_callback`.

Explicit data reception registration

```python
# Instantiate a ZigBee device object.
device = ZigBeeDevice("COM1", 9600)
device.open()

# Define callback.
def my_expl_data_received_callback(expl_xbee_message):
    address = expl_xbee_message.remote_device.get_64bit_addr()
    source_endpoint = expl_xbee_message.source_endpoint
    dest_endpoint = expl_xbee_message.dest_endpoint
    cluster = expl_xbee_message.cluster_id
    profile = expl_xbee_message.profile_id
    data = expl_xbee_message.data.decode("utf8")
    
    print("Received explicit data from %s:", data)

    # Add the callback.
device.add_expl_data_received_callback(my_expl_data_received_callback)
```

When new explicit data is received, your callback is executed providing as parameter an `ExplicitXBeeMessage` object which contains the data and other useful information:

- RemoteXBeeDevice that sent the message.
- Endpoint of the source that initiated the transmission.
- Endpoint of the destination where the message is addressed.
- Cluster ID where the data was addressed.
- Profile ID where the data was addressed.
- Byte array with the contents of the received data.
- Flag indicating if the data was sent via broadcast.
- Time when the message was received.

To stop listening to new received explicit data, use the `del_expl_data_received_callback` method to unsubscribe the already-registered callback.

Explicit data reception deregistration

```python
# [...]  
def my_expl_data_received_callback(xbee_message):
    [...]

device.add_expl_data_received_callback(my_expl_data_received_callback)
```

(continues on next page)
# Delete the callback

```python
device.del_expl_data_received_callback(my_expl_data_received_callback)
```

Example: Receive explicit data via callback

The XBee Python Library includes a sample application that demonstrates how to subscribe to the explicit data reception service in order to receive explicit data. It can be located in the following path:

```
examples/communication/explicit/ReceiveExplicitDataSample
```

**Note:** If your XBee module is configured to receive explicit data (API output mode greater than 0) and another device sends non-explicit data or a IO sample, you receive an explicit message whose application layer field values are:

- For remote data:
  - Source endpoint: 0xE8
  - Destination endpoint: 0xE8
  - Cluster ID: 0x0011
  - Profile ID: 0xC105

- For remote IO sample:
  - Source endpoint: 0xE8
  - Destination endpoint: 0xE8
  - Cluster ID: 0x0092
  - Profile ID: 0xC105

That is, when an XBee receives explicit data with these values, the message notifies the following reception callbacks in case you have registered them:

- Explicit and non-explicit data callbacks when receiving remote data.
- Explicit data callback and IO sample callback when receiving remote samples.

If you read the received data with the polling mechanism, you also receive the message through both methods.

### 2.5.6.3 Send and receive IP data

In contrast to XBee protocols like ZigBee, DigiMesh or 802.15.4, where the devices are connected each other, in cellular and Wi-Fi protocols the modules are part of the Internet.

XBee Cellular and Wi-Fi modules offer a special type of frame for communicating with other Internet-connected devices. It allows sending and receiving data specifying the destination IP address, port, and protocol (TCP, TCP SSL or UDP).

**Warning:** Only Cellular and Wi-Fi protocols support the transmission and reception of IP data. This means you cannot transmit or receive IP data using a generic `XBeeDevice` object; you must use the protocol-specific XBee device objects `CellularDevice` or `WiFiDevice`.  

### 2.5. API reference
• **Send IP data**
• **Receive IP data**

### Send IP data

IP data transmission can be a synchronous or asynchronous operation, depending on the method you use.

#### Synchronous operation

The synchronous data transmission is a blocking operation; that is, the method waits until it either receives the transmit status response or it reaches the default timeout.

The `CellularDevice` and `WiFiDevice` classes include several methods to transmit IP data synchronously:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send_ip_data(IPv4Address, Integer, IPProtocol, String or bytearray, Boolean)</code></td>
<td>Specifies the destination IP address, destination port, IP protocol (UDP, TCP or TCP SSL), data to send for transmissions and whether the socket should be closed after the transmission or not (optional).</td>
</tr>
</tbody>
</table>

#### Send network data synchronously

```python
[...]

# Instantiate a Cellular device object.
xbee = CellularDevice("COM1", 9600)
xbee.open()

# Send IP data using TCP.
dest_addr = IPv4Address("56.23.102.96")
dest_port = 5050
protocol = IPProtocol.TCP
data = "Hello XBee!"

xbee.send_ip_data(dest_addr, dest_port, protocol, data)
[...]
```

The `send_ip_data` method may fail for the following reasons:

- There is a timeout setting the IP addressing parameter, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not API or ESCAPED_API_MODE, throwing an `InvalidOperatingModeException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

#### Example: Transmit IP data synchronously

The XBee Python Library includes a sample application that demonstrates how to send IP data. You can locate the example in the following path:

`examples/communication/ip/SendIPDataSample`
Example: Transmit UDP data

The XBee Python Library includes a sample application that demonstrates how to send UDP data. You can locate the example in the following path:
examples/communication/ip/SendUDPDataSample

Example: Connect to echo server

The XBee Python Library includes a sample application that demonstrates how to connect to an echo server, send a message to it and receive its response. You can locate the example in the following path:
examples/communication/ip/ConnectToEchoServerSample

Asynchronous operation

Transmitting IP data asynchronously means that your application does not block during the transmit process. However, you cannot ensure that the data was successfully sent.

The CellularDevice and WiFiDevice classes include several methods to transmit IP data asynchronously:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_ip_data_async(IPv4Address, Integer, IPProtocol, String or bytearray, Boolean)</td>
<td>Specifies the destination IP address, destination port, IP protocol (UDP, TCP or TCP SSL), data to send for transmissions and whether the socket should be closed after the transmission or not (optional).</td>
</tr>
</tbody>
</table>

Send network data asynchronously

```python
[# Instantiate a Cellular device object.
xbree = CellularDevice("COM1", 9600)
xbree.open()  # Send IP data using TCP.
dest_addr = IPv4Address("56.23.102.96")
dest_port = 5050
protocol = IPProtocol.TCP
data = "Hello XBee!"
xbree.send_ip_data_async(dest_addr, dest_port, protocol, data)

[...]
```

The send_ip_data_async method may fail for the following reasons:

- All possible errors are caught as XBeeException:
  - The operating mode of the device is not API or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  - There is an error writing to the XBee interface, throwing a generic XBeeException.

Receive IP data

Some applications developed with the XBee Python Library may require modules to receive IP data.
XBee Cellular and Wi-Fi modules operate the same way as other TCP/IP devices. They can initiate communications with other devices or listen for TCP or UDP transmissions at a specific port. In either case, you must apply any of the receive methods explained in this section in order to read IP data from other devices.

**Listen for incoming transmissions**

If the cellular or Wi-Fi module operates as a server, listening for incoming TCP or UDP transmissions, you must start listening at a specific port, similar to the bind operation of a socket. The XBee Python Library provides a method to listen for incoming transmissions:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start_listening(Integer)</code></td>
<td>Starts listening for incoming IP transmissions in the provided port.</td>
</tr>
</tbody>
</table>

Listen for incoming transmissions

```python
# Instantiate a Cellular device object.
device = CellularDevice("COM1", 9600)
device.open()

# Listen for TCP or UDP transmissions at port 1234.
device.start_listening(1234);
```

The `start_listening` method may fail for the following reasons:

- If the listening port provided is lesser than 0 or greater than 65535, the method throws a `ValueError` error.
- If there is a timeout setting the listening port, the method throws a `TimeoutException` exception.
- Errors that register as an `XBeeException`:
  - If the operating mode of the device is not API or ESCAPED_API_MODE, the method throws an `InvalidOperatingModeException`.
  - If the response of the listening port command is not valid, the method throws an `ATCommandException`.
  - If there is an error writing to the XBee interface, the method throws a generic `XBeeException`.

You can call the `stop_listening` method to stop listening for incoming TCP or UDP transmissions:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stop_listening()</code></td>
<td>Stops listening for incoming IP transmissions.</td>
</tr>
</tbody>
</table>

Stop listening for incoming transmissions

```python
# Instantiate a Cellular device object.
device = CellularDevice("COM1", 9600)
device.open()
```
# Stop listening for TCP or UDP transmissions.
device.stop_listening()

[...]

The `stop_listening` method may fail for the following reasons:

- There is a timeout setting the listening port, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not `API` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

**Polling for IP data**

The simplest way to read IP data is by executing the `read_ip_data` method of the local Cellular or Wi-Fi devices. This method blocks your application until IP data is received or the provided timeout has expired.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_ip_data</code> <code>(integer)</code></td>
<td>Specifies the time to wait in seconds for IP data reception (method blocks during that time or until IP data is received). If you don’t specify a timeout, the method uses the default receive timeout configured in <code>XBeeDevice</code>.</td>
</tr>
</tbody>
</table>

**Read IP data (polling)**

```python
[...]
# Instantiate a Cellular device object.
device = CellularDevice("COM1", 9600)
device.open()

# Read IP data.
ip_message = device.read_ip_data()
[...]
```

The method returns the read data inside an `IPMessage` object and contains the following information:

- IP address of the device that sent the data
- Transmission protocol
- Source and destination ports
- Byte array with the contents of the received data

You can retrieve the previous information using the corresponding attributes of the `IPMessage` object:

**Get the IPMessage information**

```python
[...]
# Instantiate a cellular device object.
```
device = CellularDevice("COM1", 9600)
device.open()

# Read IP data.
ip_message = device.read_ip_data()

ip_addr = ip_message.ip_addr
source_port = ip_message.source_port
dest_port = ip_message.dest_port
protocol = ip_message.protocol
data = ip_message.data

[[...]]

You can also read IP data that comes from a specific IP address. For that purpose, the cellular and Wi-Fi device objects provide the `read_ip_data_from` method:

**Read IP data from a specific IP address (polling)**

```python
[[...]]

# Instantiate a cellular device object.
device = CellularDevice("COM1", 9600)
device.open()

# Read IP data.
ip_message = device.read_ip_data_from(IPv4Address("52.36.102.96"))
[[...]]
```

This method also returns an `IPMessage` object containing the same information described before.

**Example: Receive IP data with polling**

The XBee Python Library includes a sample application that demonstrates how to receive IP data using the polling mechanism. You can locate the example in the following path:

`examples/communication/ip/ConnectToEchoServerSample`

**IP data reception callback**

This mechanism for reading IP data does not block your application. Instead, you can be notified when new IP data has been received if you have subscribed or registered with the IP data reception service by using the `add_ip_data_received_callback` method.

**IP data reception registration**

```python
[[...]]

# Instantiate a Cellular device object.
device = CellularDevice("COM1", 9600)
device.open()

# Define the callback.

(continues on next page)
def my_ip_data_received_callback(ip_message):
    print("Received IP data from %s: %s" % (ip_message.ip_addr, ip_message.data))

# Add the callback.
device.add_ip_data_received_callback(my_ip_data_received_callback)

When new IP data is received, your callback is executed providing as parameter an IPMessage object which contains the data and other useful information:

- IP address of the device that sent the data
- Transmission protocol
- Source and destination ports
- Byte array with the contents of the received data

To stop listening to new received IP data, use the del_ip_data_received_callback method to unsubscribe the already-registered listener.

Data reception deregistration

Example: Receive IP data with listener

The XBee Python Library includes a sample application that demonstrates how to receive IP data using the listener. You can locate the example in the following path:

examples/communication/ip/ReceiveIPDataSample

2.5.6.4 Send and receive SMS messages

Another feature of the XBee Cellular module is the ability to send and receive Short Message Service (SMS) transmissions. This allows you to send and receive text messages to and from an SMS capable device such as a mobile phone.

For that purpose, these modules offer a special type of frame for sending and receiving text messages, specifying the destination phone number and data.
Warning: Only Cellular protocol supports the transmission and reception of SMS. This means you cannot send or receive text messages using a generic XBeeDevice object; you must use the protocol-specific XBee device object CellularDevice.

- Send SMS messages
- Receive SMS messages

Send SMS messages

SMS transmissions can be a synchronous or asynchronous operation, depending on the method you use.

Synchronous operation

The synchronous SMS transmission is a blocking operation; that is, the method waits until it either receives the transmit status response or it reaches the default timeout.

The CellularDevice class includes the following method to send SMS messages synchronously:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_sms(String, String)</td>
<td>Specifies the the phone number to send the SMS to and the data to send as the body of the SMS message.</td>
</tr>
</tbody>
</table>

Send SMS message synchronously

```python
([...]

# Instantiate a Cellular device object.
xbee = CellularDevice("COM1", 9600)
xbee.open()

phone_number = "+34665963205"
data = "Hello XBee!"

# Send SMS message.
xbee.send_sms(phone_number, data)

[...]```

The send_sms method may fail for the following reasons:

- If the response is not received in the configured timeout, the method throws a TimeoutException.
- If the phone number has an invalid format, the method throws a ValueError.
- Errors register as XBeeException:
  - If the operating mode of the device is not API or ESCAPED_API_MODE, the method throws an InvalidOperatingModeException.
  - If there is an error writing to the XBee interface, the method throws a generic XBeeException.
Example: Send synchronous SMS

The XBee Python Library includes a sample application that demonstrates how to send SMS messages. You can locate the example in the following path:
examples/communication/cellular/SendSMSSample

Asynchronous operation

Transmitting SMS messages asynchronously means that your application does not block during the transmit process. However, you cannot verify the SMS was successfully sent.

The `CellularDevice` class includes the following method to send SMS asynchronously:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send_sms_async(String, String)</code></td>
<td>Specifies the phone number to send the SMS to and the data to send as the body of the SMS message.</td>
</tr>
</tbody>
</table>

Send SMS message asynchronously

```python
[...]
# Instantiate a Cellular device object.
xbee = CellularDevice("COM1", 9600)
xbee.open()

phone_number = "+34665963205"
data = "Hello XBee!"

# Send SMS message.
xbee.send_sms_async(phone_number, data)
[...]
```

The `send_sms_async` method may fail for the following reasons:

- If the phone number has an invalid format, the method throws a `ValueError`.
- Errors register as `XBeeException`:
  - If the operating mode of the device is not `API` or `ESCAPED_API_MODE`, the method throws an `InvalidOperatingModeException`.
  - If there is an error writing to the XBee interface, the method throws a generic `XBeeException`.

Receive SMS messages

Some applications developed with the XBee Python Library may require modules to receive SMS messages.

**SMS reception callback**

You can be notified when a new SMS has been received if you are subscribed or registered to the SMS reception service by using the `add_sms_callback` method.

**SMS reception registration**
When a new SMS message is received, your callback is executed providing an `SMSMessage` object as a parameter. This object contains the data and the phone number that sent the message.

To stop listening to new SMS messages, use the `del_sms_callback` method to unsubscribe the already-registered listener.

Deregister SMS reception

Example: Receive SMS messages

The XBee Python Library includes a sample application that demonstrates how to subscribe to the SMS reception service in order to receive text messages. You can locate the example in the following path:

`examples/communication/cellular/ReceiveSMSSample`

### 2.5.6.5 Send and receive Bluetooth data

XBee3 modules have the ability to send and receive data from the Bluetooth Low Energy interface of the local XBee device through User Data Relay frames. This can be useful if your application wants to transmit or receive data from a cellphone connected to it over BLE.

**Warning:** Only XBee3 modules support Bluetooth Low Energy. This means that you cannot transmit or receive Bluetooth data if you don’t have one of these modules.
• Send Bluetooth data
• Receive Bluetooth data

Send Bluetooth data

The XBeeDevice class and its subclasses provide the following method to send data to the Bluetooth Low Energy interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_bluetooth_data(Bytearray)</td>
<td>Specifies the data to send to the Bluetooth Low Energy interface.</td>
</tr>
</tbody>
</table>

This method is asynchronous, which means that your application does not block during the transmit process.

Send data to Bluetooth

```python
[...]
# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()
data = "Bluetooth, are you there?"

# Send the data to the Bluetooth interface.
device.send_bluetooth_data(data.encode("utf8"))
[...]
```

The send_bluetooth_data method may fail for the following reasons:

• Errors register as XBeeException:
  – If the operating mode of the device is not API or ESCAPED_API_MODE, the method throws an InvalidOperatingModeException.
  – If there is an error writing to the XBee interface, the method throws a generic XBeeException.

Example: Send Bluetooth data

The XBee Python Library includes a sample application that demonstrates how to send data to the Bluetooth interface. You can locate the example in the following path:
examples/communication/bluetooth/SendBluetoothDataSample

Receive Bluetooth data

You can be notified when new data from the Bluetooth Low Energy interface has been received if you are subscribed or registered to the Bluetooth data reception service by using the add_bluetooth_data_received_callback method.

Bluetooth data reception registration

```python
[...]
# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
```

(continues on next page)
When a new data from the Bluetooth interface is received, your callback is executed providing the data in byte array format as parameter.

To stop listening to new data messages from the Bluetooth interface, use the `del_bluetooth_data_received_callback` method to unsubscribe the already-registered listener.

Deregister Bluetooth data reception

```
[...]
device = [...]

def my_bluetooth_data_callback(data):
    [...]

device.add_bluetooth_data_received_callback(my_bluetooth_data_callback)

[...]

def my_bluetooth_data_callback(data):
    [...]

device.add_bluetooth_data_received_callback(my_bluetooth_data_callback)

[...]

def my_bluetooth_data_callback(data):
    [...]

device.add_bluetooth_data_received_callback(my_bluetooth_data_callback)

[...]

def my_bluetooth_data_callback(data):
    [...]

device.add_bluetooth_data_received_callback(my_bluetooth_data_callback)

[...]
```

**Example: Receive Bluetooth data**

The XBee Python Library includes a sample application that demonstrates how to subscribe to the Bluetooth data reception service in order to receive data from the Bluetooth Low Energy interface. You can locate the example in the following path:

`examples/communication/bluetooth/ReceiveBluetoothDataSample`

### 2.5.6.6 Send and receive MicroPython data

XBee3 modules have the ability to send and receive data from the MicroPython interface of the local XBee device through User Data Relay frames. This can be useful if your application wants to transmit or receive data from a MicroPython program running on the module.

**Warning:** Only XBee3 and XBee Cellular modules support MicroPython. This means that you cannot transmit or receive MicroPython data if you don’t have one of these modules.

- **Send MicroPython data**
- **Receive MicroPython data**
Send MicroPython data

The XBeeDevice class and its subclasses provide the following method to send data to the MicroPython interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_micropython_data(Bytestring)</td>
<td>Specifies the data to send to the MicroPython interface.</td>
</tr>
</tbody>
</table>

This method is asynchronous, which means that your application does not block during the transmit process.

**Send data to MicroPython**

```python
[...]

# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()

data = "MicroPython, are you there?"

# Send the data to the MicroPython interface.
device.send_micropython_data(data.encode("utf8"))

[...]```

The send_micropython_data method may fail for the following reasons:

- Errors register as XBeeException:
  - If the operating mode of the device is not API or ESCAPED_API_MODE, the method throws an InvalidOperatingModeException.
  - If there is an error writing to the XBee interface, the method throws a generic XBeeException.

**Example: Send MicroPython data**

The XBee Python Library includes a sample application that demonstrates how to send data to the MicroPython interface. You can locate the example in the following path:

`examples/communication/micropython/SendMicroPythonDataSample`

Receive MicroPython data

You can be notified when new data from the MicroPython interface has been received if you are subscribed or registered to the MicroPython data reception service by using the add_micropython_data_received_callback method.

**MicroPython data reception registration**

```python
[...]

# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()

# Define the callback.
def my_micropython_data_callback(data):
    print("Data received from the MicroPython interface >> \'\$s\' % data.decode("utf-8 ...")")

(continues on next page)```
When a new data from the MicroPython interface is received, your callback is executed providing the data in byte array format as parameter.

To stop listening to new data messages from the MicroPython interface, use the \texttt{del\_micropython\_data\_received\_callback} method to unsubscribe the already-registered listener.

\textbf{Deregister MicroPython data reception}

```python
# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()
```

\textbf{Example: Receive MicroPython data}

The XBee Python Library includes a sample application that demonstrates how to subscribe to the MicroPython data reception service in order to receive data from the MicroPython interface. You can locate the example in the following path:

\texttt{examples/communication/micropython/ReceiveMicroPythonDataSample}

\section*{2.5.6.7 Receive modem status events}

A local XBee device is able to determine when it connects to a network, when it is disconnected, and when any kind of error or other events occur. The local device generates these events, and they can be handled using the XBee Python library via the modem status frames reception.

When a modem status frame is received, you are notified through the callback of a custom listener so you can take the proper actions depending on the event received.

For that purpose, you must subscribe or register to the modem status reception service using a modem status listener as parameter with the method \texttt{add\_modem\_status\_received\_callback}.

\textbf{Subscribe to modem status reception service}

```python
# Instantiate an XBee device object.
device = XBeeDevice("COM1", 9600)
device.open()
```
# Define the callback.
def my_modem_status_callback(status):
    print("Modem status: %s" % status.description)

# Add the callback.
device.add_modem_status_received_callback(my_modem_status_callback)

When a new modem status is received, your callback is executed providing as parameter a `ModemStatus` object.

To stop listening to new modem statuses, use the `del_modem_status_received_callback` method to unsubscribe the already-registered listener.

Deregister modem status

Example: Subscribe to modem status reception service

The XBee Python Library includes a sample application that shows you how to subscribe to the modem status reception service to receive modem status events. The example is located in the following path:

`examples/communication/ReceiveModemStatusSample`

## 2.5.6.8 Communicate using XBee sockets

Starting from firmware versions 13, the XBee Cellular product line includes a new set of frames to communicate with other Internet-connected devices using sockets.

The XBee Python Library provides several methods that allow you to create, connect, bind and close a socket, as well as send and receive data with it. You can use this API where the existing methods listed in the `Send and receive IP data` section limit the possibilities for an application.

**Warning:** Only the Cellular protocol supports the use of XBee sockets. This means you cannot use this API with a generic `XBeeDevice` object; you must use the protocol-specific XBee device object `CellularDevice`.

The XBee socket API is available through the `socket` class of the `digi.xbee.xsocket` module.
Create an XBee socket

Before working with an XBee socket to communicate with other devices, you have to instantiate a socket object in order to create it. To do so, you need to provide the following parameters:

- XBee Cellular device object used to work with the socket.
- IP protocol of the socket (optional). It can be IPProtocol.TCP (default), IPProtocol.UDP or IPProtocol.TCP_SSL.

```python
from digi.xbee import xsocket
from digi.xbee.devices import CellularDevice
from digi.xbee.models.protocol import IPProtocol

# Create and open an XBee Cellular device.
device = CellularDevice("COM1", 9600)  
device.open()

# Create a new XBee socket.
sock = xsocket.socket(device, IPProtocol.TCP)
```

Work with an XBee socket

Once the XBee socket is created, you can work with it to behave as a client or a server. The API offers the following methods:
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect(Tuple)</td>
<td>Connects to a remote socket at the provided address. The address must be a pair (host, port), where host is the domain name or string representation of an IPv4 and port is the numeric port value.</td>
</tr>
<tr>
<td>close()</td>
<td>Closes the socket.</td>
</tr>
<tr>
<td>bind(Tuple)</td>
<td>Binds the socket to the provided address. The address must be a pair (host, port), where host is the local interface (not used) and port is the numeric port value. The socket must not already be bound.</td>
</tr>
<tr>
<td>listen(Integer)</td>
<td>Enables a server to accept connections.</td>
</tr>
<tr>
<td>accept()</td>
<td>Accepts a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is a pair (host, port) with the address bound to the socket on the other end of the connection.</td>
</tr>
<tr>
<td>send(Bytearray)</td>
<td>Sends the provided data to the socket. The socket must be connected to a remote socket.</td>
</tr>
<tr>
<td>sendto(Bytearray, Tuple)</td>
<td>Sends the provided data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address (a pair (host, port)).</td>
</tr>
<tr>
<td>recv(Integer)</td>
<td>Receives data from the socket, specifying the maximum amount of data to be received at once. The return value is a bytearray object representing the data received.</td>
</tr>
<tr>
<td>recvfrom(Integer)</td>
<td>Receives data from the socket, specifying the maximum amount of data to be received at once. The return value is a pair (bytes, address) where bytes is a bytearray object representing the data received and address is the address of the socket sending the data (a pair (host, port)).</td>
</tr>
<tr>
<td>getsockopt(SocketOption)</td>
<td>Returns the value of the provided socket option.</td>
</tr>
<tr>
<td>setsockopt(SocketOption, Bytearray)</td>
<td>Sets the value of the provided socket option.</td>
</tr>
<tr>
<td>gettimeout()</td>
<td>Returns the configured socket timeout in seconds.</td>
</tr>
<tr>
<td>settimeout(Integer)</td>
<td>Sets the socket timeout in seconds.</td>
</tr>
<tr>
<td>getblocking()</td>
<td>Returns whether the socket is in blocking mode or not.</td>
</tr>
<tr>
<td>setblocking(Boolean)</td>
<td>Sets the socket in blocking or non-blocking mode. In blocking mode, operations block until complete or the system returns an error. In non-blocking mode, operations fail if they cannot be completed within the configured timeout.</td>
</tr>
<tr>
<td>get_sock_info()</td>
<td>Returns the information of the socket, including the socket ID, state, protocol, local port, remote port and remote address.</td>
</tr>
<tr>
<td>add_socket_state_callback(Function)</td>
<td>Adds the provided callback to be notified when a new socket state is received.</td>
</tr>
<tr>
<td>del_socket_state_callback(Function)</td>
<td>Deletes the provided socket state callback.</td>
</tr>
</tbody>
</table>

**Client sockets**

When the socket acts as a client, you just have to create and connect the socket before sending or receiving data with a remote host.

**Work with an XBee socket as client**

```python
[...]
HOST = "numbersapi.com"
PORT = "80"
```

(continues on next page)
REQUEST = "GET /random/trivia HTTP/1.1\r\nHost: numbersapi.com\r\n\r
"

# Create and open an XBee Cellular device.
device = CellularDevice("COM1", 9600)
device.open()

# Create a new XBee socket.
with xsocket.socket(device, IPProtocol.TCP) as sock:
    # Connect the socket.
    sock.connect((HOST, PORT))

    # Send an HTTP request.
    sock.send(REQUEST.encode("utf8"))

    # Receive and print the response.
    data = sock.recv(1024)
    print(data.decode("utf8"))

Example: Create a TCP client socket
The XBee Python Library includes a sample application that shows you how to create a TCP client socket to send HTTP requests. The example is located in the following path:
examples/communication/socket/SocketTCPClientSample

Server sockets

When the socket acts as a server, you must create the socket and then perform the sequence bind(), listen(), accept().

Work with an XBee socket as server

[...]
PORT = "1234"

# Create and open an XBee Cellular device.
device = CellularDevice("COM1", 9600)
device.open()

# Create a new XBee socket.
with xsocket.socket(device, IPProtocol.TCP) as sock:
    # Bind the socket to the local port.
    sock.bind((None, PORT))

    # Listen for new connections.
    sock.listen()

    # Accept new connections.
    conn, addr = sock.accept()

    with conn:
        print("Connected by %s", str(addr))
        while True:
            # Print the received data (if any).
            data = conn.recv(1024)

(continues on next page)
if data:
    print(data.decode("utf8"))

Example: Create a TCP server socket
The XBee Python Library includes a sample application that shows you how to create a TCP server socket to receive data from incoming sockets. The example is located in the following path: examples/communication/socket/SocketTCPServerSample

Example: Create a UDP server/client socket
The XBee Python Library includes a sample application that shows how to create a UDP socket to deliver messages to a server and listen for data coming from multiple peers. The example is located in the following path: examples/communication/socket/SocketUDPServerClientSample

2.5.7 Handle analog and digital IO lines
All the XBee modules, regardless of the protocol they run, have a set of lines (pins). You can use these pins to connect sensors or actuators and configure them with specific behavior.

You can configure the IO lines of an XBee device to be digital input/output (DIO), analog to digital converter (ADC), or pulse-width modulation output (PWM). The configuration you provide to a line depends on the device where you want to connect.

Note: All the IO management features displayed in this topic and sub-topics are applicable for both local and remote XBee devices.

The XBee Python Library exposes an easy way to configure, read, and write the IO lines of the local and remote XBee devices through the following corresponding classes:

- XBeeDevice for local devices.
- RemoteXBeeDevice for remotes.

2.5.7.1 Configure the IO lines
All XBee device objects include a configuration method, set_io_configuration(), where you can specify the IO line being configured and the desired function being set.

For the IO line parameter, the API provides an enumerator called IOLine that helps you specify the desired IO line easily by functional name. This enumerator is used along all the IO related methods in the API.

The supported functions are also contained in an enumerator called IOMode. You can choose between the following functions:

- DISABLED
- SPECIAL_FUNCTIONALITY (Shouldn’t be used to configure IOs)
- PWM
- ADC
- DIGITAL_IN
- DIGITAL_OUT_LOW
• DIGITAL_OUT_HIGH

Configure local or remote IO lines

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Instantiate a remote XBee device object.
remote_xbee = RemoteXBeeDevice(local_xbee, XBee64BitAddress.from_hex_string("0013A20012345678"))

# Configure the DIO1_AD1 line of the local device to be Digital output (set high by default).
local_xbee.set_io_configuration(IOLine.DIO1_AD1, IOMode.DIGITAL_OUT_HIGH)

# Configure the DIO2_AD2 line of the local device to be Digital input.
local_xbee.set_io_configuration(IOLine.DIO2_AD2, IOMode.DIGITAL_IN)

# Configure the DIO3_AD3 line of the remote device to be Analog input (ADC).
remote_xbee.set_io_configuration(IOLine.DIO3_AD3, IOMode.ADC)

# Configure the DIO10_PWM0 line of the remote device to be PWM output (PWM).
remote_xbee.set_io_configuration(IOLine.DIO10_PWM0, IOMode.PWM)
```

The `set_io_configuration()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

You can read the current configuration of any IO line the same way an IO line can be configured with a desired function using the corresponding getter, `get_io_configuration()`.

Get IO configuration

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Get the configuration mode of the DIO1_AD1 line.
io_mode = local_xbee.get_io_configuration(IOLine.DIO1_AD1)
```

The `get_io_configuration()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`. 

Chapter 2. Contents
• Other errors caught as XBeeException:
  – The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  – The response of the command is not valid, throwing an ATCommandException.
  – There is an error writing to the XBee interface, throwing a generic XBeeException.

Digital Input/Output

If your IO line is configured as digital output, you can set its state (high/low) easily. All the XBee device classes provide the method, `set_dio_value()`, with the desired IOLine as the first parameter and an IOValue as the second. The IOValue enumerator includes HIGH and LOW as possible values.

Set digital output values

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Set the DIO2_AD2 line low.
local_xbee.set_dio_value(IOLine.DIO2_AD2, IOValue.LOW)

# Set the DIO2_AD2 line high.
local_xbee.set_dio_value(IOLine.DIO2_AD2, IOValue.HIGH)
```

The `set_dio_value()` method may fail for the following reasons:

• ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.

• Other errors caught as XBeeException:
  – The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  – The response of the command is not valid, throwing an ATCommandException.
  – There is an error writing to the XBee interface, throwing a generic XBeeException.

You can also read the current status of the pin (high/low) by issuing the method `get_dio_value()`. The parameter of the method must be the IO line to be read.

Read digital input values

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Get the value of the DIO2_AD2.
value = local_xbee.get_dio_value(IOLine.DIO2_AD2)
```

The `get_dio_value()` method may fail for the following reasons:
• ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
• Other errors caught as `XBeeException`:
  – The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  – If the received response does not contain the value for the given IO line, throwing an `OperationNotSupportedException`. This can happen (for example) if you try to read the DIO value of an IO line that is not configured as DIO.
  – The response of the command is not valid, throwing an `ATCommandException`.
  – There is an error writing to the XBee interface, throwing a generic `XBeeException`.

Example: Handle DIO IO lines

The XBee Python Library includes two sample applications that demonstrate how to handle DIO lines in your local and remote XBee Devices. The examples are located in the following path:

- `examples/io/LocalDIOSample/LocalDIOSample.py`
- `examples/io/RemoteDIOSample/RemoteDIOSample.py`

ADC

When you configure an IO line as analog to digital converter (ADC), you can only read its value (counts) with `get_adc_value()`. In this case, the method used to read ADCs is different than the digital I/O method, but the parameter provided is the same: the IO line to read the value from.

Read ADC values

```python
[...]
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()
[...]
# Get the value of the DIO 3 (analog to digital converter).
value = local_xbee.get_adc_value(IOLine.DIO3_AD3)
[...]
```

The `get_adc_value()` method may fail for the following reasons:

• ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
• Other errors caught as `XBeeException`:
  – The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  – If the received response does not contain the value for the given IO line, throwing an `OperationNotSupportedException`. This can happen (for example) if you try to read the ADC value of an IO line that is not configured as ADC.
  – The response of the command is not valid, throwing an `ATCommandException`.
  – There is an error writing to the XBee interface, throwing a generic `XBeeException`. 

80 Chapter 2. Contents
Example: Handle ADC IO lines

The XBee Python Library includes two sample applications that demonstrate how to handle ADC lines in your local and remote XBee devices. The examples are located in the following path:

- examples/io/LocalADCSample/LocalADCSample.py
- examples/io/RemoteADCSample/RemoteADCSample.py

PWM

Not all the XBee protocols support pulse-width modulation (PWM) output handling, but the XBee Python Library provides functionality to manage them. When you configure an IO line as PWM output, you must use specific methods to set and read the duty cycle of the PWM.

For the set case, use the method set_pwm_duty_cycle() and provide the IO line configured as PWM and the value of the duty cycle in % of the PWM. The duty cycle is the proportion of ‘ON’ time to the regular interval or ‘period’ of time. A high duty cycle corresponds to high power, because the power is ON for most of the time. The percentage parameter of the set duty cycle method is a double, which allows you to be more precise in the configuration.

Set the duty cycle of an IO line configure as PWM

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Set a duty cycle of 75% to the DIO10_PWM0 line (PWM output).
local_xbee.set_pwm_duty_cycle(IOLine.DIO10_PWM0, 75)
```

The set_pwm_duty_cycle() method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.
- Other errors caught as XBeeException:
  - The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  - The response of the command is not valid, throwing an ATCommandException.
  - There is an error writing to the XBee interface, throwing a generic XBeeException.

Get the duty cycle of an IO line configured as PWM

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()
```

(continues on next page)
# Get the duty cycle of the DIO10_PWM0 line (PWM output).

```python
duty_cycle = local_xbee.get_pwm_duty_cycle(IOLine.DIO10_PWM0);
```

[...]

**Note:** In both cases (get and set), the IO line provided must be PWM capable and must be configured as PWM output.

## 2.5.7.2 Read IO samples

XBee modules can monitor and sample the analog and digital IO lines. You can read IO samples locally or transmitted to a remote device to provide an indication of the current IO line states.

There are three ways to obtain IO samples on a local or remote device:

- Queried sampling
- Periodic sampling
- Change detection sampling

The XBee Python Library represents an IO sample by the `IOSample` class, which contains:

- Digital and analog channel masks that indicate which lines have sampling enabled.
- Values of those enabled lines.

You must configure the IO lines you want to receive in the IO samples before enabling sampling.

### Queried sampling

The XBee Python Library provides a method to read an IO sample that contains all enabled digital IO and analog input channels, `read_io_sample()`. The method returns an `IOSample` object.

### Read an IO sample and getting the DIO value

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Read an IO sample from the device.
io_sample = local_xbee.read_io_sample()

# Select the desired IO line.
io_line = IOLine.DIO3_AD3

# Check if the IO sample contains the expected IO line and value.
if io_sample.has_digital_value(io_line):
    print("DIO3 value: ", io_sample.get_digital_value(io_line))
```

[...]
The `read_io_sample()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

**Periodic sampling**

Periodic sampling allows an XBee module to take an IO sample and transmit it to a remote device at a periodic rate. That remote device is defined in the destination address through the `set_dest_address()` method. The XBee Python Library provides the `set_io_sampling_rate()` method to configure the periodic sampling.

The XBee module samples and transmits all enabled digital IO and analog inputs to the remote device every X seconds. A sample rate of 0 s disables this feature.

**Set the IO sampling rate**

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Set the destination address.
local_xbee.set_dest_address(XBee64BitAddress.from_hex_string("0013A20040XXXXXX"))

# Set the IO sampling rate.
local_xbee.set_io_sampling_rate(5)  # 5 seconds.
```

The `set_io_sampling_rate()` method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a `TimeoutException`.
- Other errors caught as `XBeeException`:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBee interface, throwing a generic `XBeeException`.

You can also read this value using the `get_io_sampling_rate()` method. This method returns the IO sampling rate in milliseconds and ‘0’ when the feature is disabled.

**Get the IO sampling rate**

```python
# Instantiate an XBee device object.
```
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Get the IO sampling rate.
value = local_xbee.get_io_sampling_rate()

The `get_io_sampling_rate()` method may fail for the following reasons:

- **ACK of the command sent is not received in the configured timeout**, throwing a `TimeoutException`.
- **Other errors caught as XBeeException**:
  - The operating mode of the device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.
  - The response of the command is not valid, throwing an `ATCommandException`.
  - There is an error writing to the XBees interface, throwing a generic `XBeeException`.

### 2.5.7.3 Change detection sampling

You can configure modules to transmit a data sample immediately whenever a monitored digital IO pin changes state. The `set_dio_change_detection()` method establishes the set of digital IO lines that are monitored for change detection. A `None` set disables the change detection sampling.

As in the periodic sampling, change detection samples are transmitted to the configured destination address.

**Note:** This feature only monitors and samples digital IOs, so it is not valid for analog lines.

#### Set the DIO change detection

```python
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Set the destination address.
local_xbee.set_dest_address(XBee64BitAddress.from_hex_string("0013A20040XXXXXX"))

# Create a set of IO lines to be monitored.
lines = [IOLine.DIO3_AD3, IOLine.DIO4_AD4]

# Enable the DIO change detection sampling.
local_xbee.set_dio_change_detection(lines)
```

The `set_dio_change_detection()` method may fail for the following reasons:

- **ACK of the command sent is not received in the configured timeout**, throwing a `TimeoutException`. 
Other errors caught as XBeeException:

- The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
- The response of the command is not valid, throwing an ATCommandException.
- There is an error writing to the XBee interface, throwing a generic XBeeException.

You can also get the lines that are monitored using the get_dio_change_detection() method. A None value indicates that this feature is disabled.

Get the DIO change detection

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Get the set of lines that are monitored.
lines = local_xbee.get_dio_change_detection()

# The get_dio_change_detection() method may fail for the following reasons:
- ACK of the command sent is not received in the configured timeout, throwing a TimeoutException.
- Other errors caught as XBeeException:
  - The operating mode of the device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.
  - The response of the command is not valid, throwing an ATCommandException.
  - There is an error writing to the XBee interface, throwing a generic XBeeException.
```

Register an IO sample listener

In addition to configuring an XBee device to monitor and sample the analog and digital IO lines, you must register a callback in the local device where you want to receive the IO samples. You are then notified when the device receives a new IO sample.

You must subscribe to the IO samples reception service by using the method add_io_sample_received_callback() with an IO sample reception callback function as parameter.

Add an IO sample callback

```python
# Define the IO sample receive callback.

(continues on next page)```
This callback function will receive three parameters when an IO sample receive event is raised:

- The received IO sample as an `IOSample` object.
- The remote XBee device that sent the IO sample as a `RemoteXBeeDevice` object.
- The time in which the IO sample was received as an `Float` (calculated with Python standard `time.time()`).

To stop receiving notifications of new IO samples, remove the added callback using the `del_io_sample_received_callback()` method.

### Remove an IO sample callback

```python
# Instantiate an XBee device object.
local_xbee = XBeeDevice("COM1", 9600)
local_xbee.open()

# Define the IO sample receive callback.
def io_sample_callback(io_sample, remote_xbee, send_time):
    print("IO sample received at time \$s." % str(send_time))
    print("IO sample:")
    print(str(io_sample))

# Subscribe to IO samples reception by adding the callback.
local_xbee.add_io_sample_received_callback(io_sample_callback)

# Unsubscribe from IO samples reception by removing the callback.
local_xbee.del_io_sample_received_callback(io_sample_callback)
```

The `del_io_sample_received_callback()` method will raise a `ValueError` if you try to delete a callback that you have not added yet.

### Example: Receive IO samples

The XBee Python Library includes a sample application that demonstrates how to configure a remote device to monitor IO lines and receive the IO samples in the local device. The example is located in the following path:

`examples/io/IOSamplingSample/IOSamplingSample.py`
2.5.8 Update the XBee

To keep your XBee devices up to date, the XBee Python Library provides several methods to update the device software including firmware, file system and XBee profiles:

- Update the XBee firmware
- Update the XBee file system
- Apply an XBee profile

**Warning:** At the moment, firmware, file system, and profile updates are only supported in **XBee 3** devices.

### 2.5.8.1 Update the XBee firmware

You may need to update the running firmware of your XBee devices to, for example, change their XBee protocol, fix issues and security risks, or access to new features and functionality.

The XBee Python Library provides methods to perform firmware updates in local and remote devices:

- Update the firmware of a local XBee
- Update the firmware of a remote XBee

**Warning:** At the moment, firmware update is only supported in **XBee 3** devices.

#### Update the firmware of a local XBee

The firmware update process of a local XBee device is performed over the serial connection. For this operation, you need the following components:

- The XBee device object instance or the serial port name where the device is attached to.
- The new firmware XML descriptor file.
- The new firmware binary file (*.gbl)
- Optionally, the new bootloader binary file (*.gbl) required by the new firmware.

**Warning:** Firmware update will fail if the firmware requires a new bootloader and it is not provided.

**Example: Local Firmware Update**

The XBee Python Library includes a sample application that displays how to perform a local firmware update. It can be located in the following path:

`examples/firmware/LocalFirmwareUpdateSample/LocalFirmwareUpdateSample.py`

#### Update the local firmware using an XBee device object

If you have an object instance of your local XBee device, you have to call the `update_firmware` method of the `XBeeDevice` class providing the required parameters:
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>update_firmware(String, String, String, Integer, Function)</td>
<td>Performs a firmware update operation of the device.</td>
</tr>
</tbody>
</table>

- **xml_firmware_file (String):** path of the XML file that describes the firmware to upload.
- **xbee_firmware_file (String, optional):** location of the XBee binary firmware file (*.gbl).
- **bootloader_firmware_file (String, optional):** location of the bootloader binary firmware file (*.gbl).
- **timeout (Integer, optional):** the maximum amount of seconds to wait for target read operations during the update process.
- **progress_callback (Function, optional):** function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

The `update_firmware` method may fail for the following reasons:

- The device does not support the firmware update operation, throwing a `OperationNotSupportedException`.
- There is an error during the firmware update operation, throwing a `FirmwareUpdateException`.
- Other errors caught as `XBeeException`:
  - The device is not open, throwing a generic `XBeeException`.
  - The operating mode of the local XBee device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.

Update local XBee device firmware using an XBee device object

```python
[...]
XML_FIRMWARE_FILE = "my_path/my_firmware.xml"
XBEE_FIRMWARE_FILE = "my_path/my_firmware.gbl"
BOOTLOADER_FIRMWARE_FILE = "my_path/my_bootloader.gbl"
[...]
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  
[...]
# Update the XBee device firmware.
device.update_firmware(XML_FIRMWARE_FILE,
                        xbee_firmware_file=XBEE_FIRMWARE_FILE,
                        bootloader_firmware_file=BOOTLOADER_FIRMWARE_FILE,
                        progress_callback=progress_callback,)
[...]
```
Update the local firmware using a serial port

If you do not know the XBee serial communication parameters or you cannot instantiate the XBee device object (for example if the device must be recovered), you can perform the firmware update process by providing the serial port identifier where the XBee is attached to.

In this scenario, use the `update_local_firmware` method of the `XBee firmware` module providing the required parameters. The library forces the XBee to reboot into bootloader mode, using the recovery mechanism, and performs the firmware update from that point.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>update_local_firmware(String or XBeeDevice, String, String, String, Integer, Function)</code></td>
<td>Performs a local firmware update operation in the given target.</td>
</tr>
<tr>
<td>• `xml_firmware_file (String): path of the XML file that describes the firmware to upload.</td>
<td></td>
</tr>
<tr>
<td>• `xbee_firmware_file (String, optional): location of the XBee binary firmware file (*.gbl).</td>
<td></td>
</tr>
<tr>
<td>• `bootloader_firmware_file (String, optional): location of the bootloader binary firmware file.</td>
<td></td>
</tr>
<tr>
<td>• `timeout (Integer, optional): the maximum amount of seconds to wait for target read operations during the update process.</td>
<td></td>
</tr>
<tr>
<td>• <code>progress_callback (Function, optional): function to execute to receive progress information. Receives two arguments: – The current update task as a String – The current update task percentage as an Integer</code></td>
<td></td>
</tr>
</tbody>
</table>

The `update_local_firmware` method may fail for the following reasons:

• There is an error during the firmware update operation, throwing a `FirmwareUpdateException`.

Update local XBee device firmware using the serial port name

```python
import digi.xbee.firmware

[...]
SERIAL_PORT = "COM1"

XML_FIRMWARE_FILE = "my_path/my_firmware.xml"
XBEE_FIRMWARE_FILE = "my_path/my_firmware.gbl"
BOOTLOADER_FIRMWARE_FILE = "my_path/my_bootloader.gbl"

[...]
# Update the XBee device firmware using the serial port name.
firmware.update_local_firmware(SERIAL_PORT,
        XML_FIRMWARE_FILE,
```
Update the firmware of a remote XBee

The firmware update process for remote XBee devices is performed over the air using special XBee frames. For this operation, you need the following components:

- The remote XBee device object instance.
- The new firmware XML descriptor file.
- The new firmware binary file (*.ota)
- Optionally, the new firmware binary file with the bootloader embedded (*.otb)

**Warning:** Firmware update fails if the firmware requires a new bootloader and the *.otb file is not provided.

To perform the remote firmware update, call the `update_firmware` method of the `RemoteXBeeDevice` class providing the required parameters:

```
update_firmware(xml_firmware_file=XBEE_FIRMWARE_FILE,
                xbee_firmware_file=XBEE_FIRMWARE_FILE,
                bootloader_firmware_file=BOOTLOADER_FIRMWARE_FILE,
                progress_callback=progress_callback,)
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>update_firmware(String, String, String, Integer, Function)</code></td>
<td>Performs a remote firmware update operation of the device.</td>
</tr>
<tr>
<td></td>
<td>• <code>xml_firmware_file (String)</code>: path of the XML file that describes the firmware to upload.</td>
</tr>
<tr>
<td></td>
<td>• <code>xbee_firmware_file (String, optional)</code>: location of the XBee binary firmware file (*.ota).</td>
</tr>
<tr>
<td></td>
<td>• <code>bootloader_firmware_file (String, optional)</code>: location of the XBee binary firmware file with bootloader embedded (*.otb).</td>
</tr>
<tr>
<td></td>
<td>• <code>timeout (Integer, optional)</code>: the maximum amount of seconds to wait for target read operations during the update process.</td>
</tr>
<tr>
<td></td>
<td>• <code>progress_callback (Function, optional)</code>: function to execute to receive progress information. Receives two arguments:</td>
</tr>
<tr>
<td></td>
<td>– The current update task as a String</td>
</tr>
<tr>
<td></td>
<td>– The current update task percentage as an Integer</td>
</tr>
</tbody>
</table>

The `update_firmware` method may fail for the following reasons:

- The remote device does not support the firmware update operation, throwing a `OperationNotSupportedException`.
- There is an error during the firmware update operation, throwing a `FirmwareUpdateException`.
- Other errors caught as `XBeeException`:
– The local device is not open, throwing a generic XBeeException.
– The operating mode of the local device is not API_MODE or ESCAPED_API_MODE, throwing an InvalidOperatingModeException.

Update remote XBee device firmware

```python
...
XML_FIRMWARE_FILE = "my_path/my_firmware.xml"
OTA_FIRMWARE_FILE = "my_path/my_firmware.ota"
OTB_FIRMWARE_FILE = "my_path/my_firmware.otb"
REMOTE_DEVICE_NAME = "REMOTE"
...

# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

# Get the network.
xnet = xbee.get_network()  

# Get the remote device.
remote = xnet.discover_device(REMOTE_DEVICE_NAME)  

# Update the remote XBee device firmware.
remote.update_firmware(SERIAL_PORT,
           XML_FIRMWARE_FILE,
           xbee_firmware_file=OTA_FIRMWARE_FILE,
           bootloader_firmware_file=OTB_FIRMWARE_FILE,
           progress_callback=progress_callback,)  

[...]
```

Example: Remote Firmware Update

The XBee Python Library includes a sample application that displays how to perform a remote firmware update. It can be located in the following path:
`examples/firmware/RemoteFirmwareUpdateSample/RemoteFirmwareUpdateSample.py`

2.5.8.2 Update the XBee file system

XBee 3 devices feature file system capabilities, meaning that they are able to persistently store files and folders in flash. The XBee Python Library provides classes and methods to manage these files.

- Create file system manager
- File system operations

**Warning:** At the moment file system capabilities are only supported in local XBee 3 devices.
Create file system manager

A `LocalXBeeFileSystemManager` object is required to work with local devices file system. You can instantiate this class by providing the local XBee device object. Once you have the object instance, you must call the `connect` method to open the file system connection and leave it ready to work.

**Warning:** File system operations take ownership of the serial port, meaning that you will stop receiving messages from the device until file system connection is closed. For this reason it is highly recommended to call the `disconnect` method of the file system manager as soon as you finish working with it.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connect()</code></td>
<td>Connects the file system manager.</td>
</tr>
<tr>
<td><code>disconnect()</code></td>
<td>Disconnects the file system manager and restores the device connection.</td>
</tr>
</tbody>
</table>

The `connect` method may fail for the following reasons:

- The device does not support the file system capabilities, throwing a `FileSystemNotSupportedException`.
- There is an error during the connect operation, throwing a `FileSystemException`.

Create a local file system manager

```python
from digi.xbee.filesystem import LocalXBeeFileSystemManager

[...]

# Instantiate an XBee device object.
xbee = XBeeDevice(...)

[...]

# Create the file system manager and connect it.
filesystem_manager = LocalXBeeFileSystemManager(device)
filesystem_manager.connect()

[...]

filesystem_manager.disconnect()
[...]
```

File system operations

The file system manager provides several methods to navigate through the device file system and operate with the different files and folders:
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_current_directory()</td>
<td>Returns the current device directory.</td>
</tr>
<tr>
<td>change_directory(String)</td>
<td>Changes the current device working directory to the given one.</td>
</tr>
<tr>
<td></td>
<td>• directory (String): the new directory to change to.</td>
</tr>
<tr>
<td>make_directory(String)</td>
<td>Creates the provided directory.</td>
</tr>
<tr>
<td></td>
<td>• directory (String): the new directory to create.</td>
</tr>
<tr>
<td>list_directory(String)</td>
<td>Lists the contents of the given directory.</td>
</tr>
<tr>
<td></td>
<td>• directory (String, optional): the directory to list its contents. Optional.</td>
</tr>
<tr>
<td></td>
<td>If not provided, the current directory contents are listed.</td>
</tr>
<tr>
<td>remove_element(String)</td>
<td>Removes the given file system element path.</td>
</tr>
<tr>
<td></td>
<td>• element_path (String): path of the file system element to remove.</td>
</tr>
<tr>
<td>move_element(String, String)</td>
<td>Moves the given source element to the given destination path.</td>
</tr>
<tr>
<td></td>
<td>• source_path (String): source path of the element to move.</td>
</tr>
<tr>
<td></td>
<td>• dest_path (String): destination path of the element to move.</td>
</tr>
<tr>
<td>put_file(String, String, Boolean, Function)</td>
<td>Transfers the given file in the specified destination path of the XBee device.</td>
</tr>
<tr>
<td></td>
<td>• source_path (String): the path of the file to transfer.</td>
</tr>
<tr>
<td></td>
<td>• dest_path (String): the destination path to put the file in.</td>
</tr>
<tr>
<td></td>
<td>• secure (Boolean, optional): True if the file should be stored securely. False otherwise. Defaults to False.</td>
</tr>
<tr>
<td></td>
<td>• progress_callback (Function, optional): function to execute to receive progress information. Takes the following arguments:</td>
</tr>
<tr>
<td></td>
<td>– The progress percentage as integer.</td>
</tr>
<tr>
<td>put_dir(String, String, Function)</td>
<td>Uploads the given source directory contents into the given destination directory in the device.</td>
</tr>
<tr>
<td></td>
<td>• source_dir (String): the local directory to upload its contents.</td>
</tr>
<tr>
<td></td>
<td>• dest_dir (String, optional): the remote directory to upload the contents to. Defaults to current directory.</td>
</tr>
<tr>
<td></td>
<td>• progress_callback (Function, optional): function to execute to receive progress information. Takes the following arguments:</td>
</tr>
<tr>
<td></td>
<td>– The file being uploaded as string.</td>
</tr>
<tr>
<td></td>
<td>– The progress percentage as integer.</td>
</tr>
<tr>
<td>get_file(String, String, Function)</td>
<td>Downloads the given XBee device file in the specified destination path.</td>
</tr>
<tr>
<td></td>
<td>• source_path (String): the path of the XBee device file to download.</td>
</tr>
<tr>
<td></td>
<td>• dest_path (String): the destination path to store the file in.</td>
</tr>
</tbody>
</table>

2.5. API reference
The methods above may fail for the following reasons:

- There is an error executing the requested operation, throwing a FileSystemException.

### Example: Format file system

The XBee Python Library includes a sample application that displays how to format the device file system. It can be located in the following path:

```python
examples/filesystem/FormatFilesystemSample/FormatFilesystemSample.py
```

### Example: List directory

The XBee Python Library includes a sample application that displays how to list the contents of a device directory. It can be located in the following path:

```python
examples/filesystem/ListDirectorySample/ListDirectorySample.py
```

### Example: Upload/download file

The XBee Python Library includes a sample application that displays how to upload/download a file from the device. It can be located in the following path:

```python
examples/filesystem/UploadDownloadFileSample/UploadDownloadFileSample.py
```

### 2.5.8.3 Apply an XBee profile

An XBee profile is a snapshot of a specific XBee configuration, including firmware, settings, and file system contents. The XBee Python API includes a set of classes and methods to work with XBee profiles and apply them to local and remote devices.

- Read an XBee profile
- Apply an XBee profile to a local device
- Apply an XBee profile to a remote device

To configure individual settings see Configure the XBee device.

**Note:** Use XCTU to create configuration profiles.

**Warning:** At the moment applying profiles is only supported in XBee 3 devices.

### Read an XBee profile

The library provides a class called `XBeeProfile` that is used to read and extract information of an existing XBee profile file.

To create a `XBeeProfile` object, provide the location of the profile file in the class constructor.

#### Instantiate an XBee profile

```python
from digi.xbee.profile import XBeeProfile

[...]
```
The creation of the XBee profile object may fail for the following reasons:

- The provided profile file is not valid, throwing a `ValueError`.
- There is any error reading the profile file, throwing a `ProfileReadException`.

Once the XBee profile object is created, you can extract all the profile information by accessing each of the exposed properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile_file</td>
<td>Returns the profile file.</td>
</tr>
<tr>
<td>version</td>
<td>Returns the profile version.</td>
</tr>
<tr>
<td>flash_firmware_option</td>
<td>Returns the profile flash firmware option.</td>
</tr>
<tr>
<td>description</td>
<td>Returns the profile description.</td>
</tr>
<tr>
<td>reset_settings</td>
<td>Returns whether the settings of the XBee device are reset before applying the profile ones or not.</td>
</tr>
<tr>
<td>has_filesystem</td>
<td>Returns whether the profile has filesystem information or not.</td>
</tr>
<tr>
<td>profile_settings</td>
<td>Returns all the firmware settings that the profile configures.</td>
</tr>
<tr>
<td>firmware_version</td>
<td>Returns the compatible firmware version of the profile.</td>
</tr>
<tr>
<td>hardware_version</td>
<td>Returns the compatible hardware version of the profile.</td>
</tr>
<tr>
<td>firmware_description_file</td>
<td>Returns the path of the profile firmware description file.</td>
</tr>
<tr>
<td>file_system_path</td>
<td>Returns the profile file system path.</td>
</tr>
</tbody>
</table>

Read an XBee profile

```python
from digi.xbee.profile import XBeeProfile

PROFILE_PATH = "~/home/user/my_profile.xpro"

# Create the XBee profile object.
xbee_profile = XBeeProfile(PROFILE_PATH)

# Print profile compatible hardware and software versions
print(" - Firmware version: " % xbee_profile.firmware_version)
print(" - Hardware version: " % xbee_profile.hardware_version)
```

Example: Read an XBee profile

The XBee Python Library includes a sample application that displays how to read an XBee profile. It can be located in the following path:

```
examples/profile/ReadXBeeProfileSample/ReadXBeeProfileSample.py
```
Apply an XBee profile to a local device

Applying a profile to a local XBee device requires the following components:

- The local XBee device object instance.
- The profile file to apply (*.xpro).

**Note:** Use XCTU to create configuration profiles.

To apply the XBee profile to a local XBee, you have to call the `apply_profile` method of the `XBeeDevice` class providing the required parameters:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| `apply_profile(String, Function)` | Applies the given XBee profile to the XBee device.  
  - **profile_path (String):** path of the XBee profile file to apply.  
  - **progress_callback (Function, optional):** function to execute to receive progress information. Receives two arguments:
    - The current apply profile task as a String
    - The current apply profile task percentage as an Integer |

The `apply_profile` method may fail for the following reasons:

- The local device does not support the apply profile operation, throwing an `OperationNotSupportedException`.
- There is an error while applying the XBee profile, throwing a `UpdateProfileException`.
- Other errors caught as `XBeeException`:
  - The local device is not open, throwing a generic `XBeeException`.
  - The operating mode of the local device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.

Apply an XBee profile to a local device

```python
PROFILE_PATH = "/home/user/my_profile.xpro"

# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

# Apply the XBee device profile.
device.apply_profile(PROFILE_PATH, progress_callback=progress_callback)
```
Example: Apply local XBee profile

The XBee Python Library includes a sample application that displays how to apply an XBee profile to a local device. It can be located in the following path:
examples/profile/ApplyXBeeProfileSample/ApplyXBeeProfileSample.py

Apply an XBee profile to a remote device

Applying a profile to a remote XBee requires the following components:

- The remote XBee device object instance.
- The profile file to apply (*.xpro).

Note: Use XCTU to create configuration profiles.

To apply the XBee profile to a remote XBee device, you have to call the `apply_profile` method of the `RemoteXBeeDevice` class providing the required parameters:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply_profile(String, Function)</code></td>
<td>Applies the given XBee profile to the remote XBee device.</td>
</tr>
<tr>
<td></td>
<td>• <code>profile_path</code> (String): path of the XBee profile file to apply.</td>
</tr>
<tr>
<td></td>
<td>• <code>progress_callback</code> (Function, optional): function to execute to</td>
</tr>
<tr>
<td></td>
<td>receive progress information. Receives two arguments:</td>
</tr>
<tr>
<td></td>
<td>– The current apply profile task as a String</td>
</tr>
<tr>
<td></td>
<td>– The current apply profile task percentage as an Integer</td>
</tr>
</tbody>
</table>

The `apply_profile` method may fail for the following reasons:

- The remote device does not support the apply profile operation, throwing a `OperationNotSupportedException`.
- There is an error while applying the XBee profile, throwing a `UpdateProfileException`.
- Other errors caught as `XBeeException`:
  - The local device is not open, throwing a generic `XBeeException`.
  - The operating mode of the local device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.

Apply an XBee profile to a remote device

```python
[...
PROFILE_PATH = "~/home/user/my_profile.xpro"
REMOTE_DEVICE_NAME = "REMOTE"
[...
```

(continues on next page)
# Instantiate an XBee device object.
xbee = XBeeDevice(...)  

# Get the network.
xnet = xbee.get_network()  

# Get the remote device.
remote = xnet.discover_device(REMOTE_DEVICE_NAME)  

[...]  

# Apply the XBee device profile.
remote.apply_profile(PROFILE_PATH, progress_callback=progress_callback)  

[...]  

Example: Apply remote XBee profile

The XBee Python Library includes a sample application that displays how to apply an XBee profile to a remote
device. It can be located in the following path:
examples/profile/ApplyXBeeProfileRemoteSample/ApplyXBeeProfileRemoteSample.py

2.5.9 Log events

Logging is a fundamental part of applications, and every application includes this feature. A well-designed logging
system is a useful utility for system administrators, developers, and the support team and can save valuable time in
sorting through the cause of issues. As users execute programs on the front end, the system invisibly builds a vault of
event information (log entries).

The XBee Python Library uses the Python standard logging module for registering logging events. The logger works
at module level; that is, each module has a logger with a unique name.

The modules that have logging integrated are devices and reader. By default, all loggers are disabled so you will
not see any logging message in the console if you do not activate them.

In the XBee Python Library, you need three things to enable the logger:

1. The logger itself.
2. A handler. This will determine if the messages will be displayed in the console, written in a file, sent through a
   socket, etc.
3. A formatter. This will determine the message format. For example, a format could be:
   
   • Timestamp with the current date - logger name - level (debug, info, warning...) - data.

To retrieve the logger, use the get_logger() method of the logging module, providing the name of the logger that
you want to get as parameter. In the XBee Python Library all loggers have the name of the module they belong to. For
example, the name of the logger of the devices module is digi.xbee.devices. You can get a module name
with the special attribute \_\_name\_\_.

Retrieve a module name and its logger

```python
import logging

[...]
```
To retrieve a handler, you can use the default Python handler or create your own one. Depending on which type of handler you use, the messages created by the logger will be printed in the console, in a file, etc. You can have more than one handler per logger, this means that you can enable the default XBee Python Library handler and add your own handlers.

**Retrieve a handler and add it to a logger**

```python
import logging

# Get the logger of the devices module.
dev_logger = logging.getLogger(digi.xbee.devices.__name__)

# Get a handler and add it to the logger.
handler = logging.StreamHandler()
dev_logger.addHandler(handler)
```

The previous code snippet shows how to add a handler to a logger, but the logical way is to add a formatter to a handler, and then add the handler to the logger.

When you create a formatter, you must specify which information will be printed and in which format. This guide shows you how to create a formatter with a simple format. If you want to create more complex formatters or handlers, see the Python documentation.

**Create a formatter and add it to a handler**

```python
import logging

# Get a handler.
handler = (...)  

# Instantiate a formatter so the log entries are represented as defined here.
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - '  
'    %(message)s')

# Configure the formatter in the handler.
handler.setFormatter(formatter)
```

**Enable a logger for the devices module**

```python
import logging
```
# Get the logger of the devices module providing the name.
dev_logger = logging.getLogger("digi.xbee.devices")

# Get a handler and configure a formatter for it.
handler = logging.StreamHandler()
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)

# Add the handler to the logger.
dev_logger.addHandler(handler)

2.5.9.1 Logging level

The XBee Python Library also provides a method in the `utils` module, `enable_logger()`, to enable the logger with the default settings. These settings are:

- **Handler**: StreamHandler
- **Format**: timestamp - logger name - level - message

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enable_logger(name, level=logging.DEBUG)</code></td>
<td>Enables the logger.</td>
</tr>
<tr>
<td></td>
<td>• name: the name of the module whose logger you want to activate.</td>
</tr>
<tr>
<td></td>
<td>• level: default DEBUG. The level you want to see.</td>
</tr>
</tbody>
</table>

Enable a logger

```python
import logging

# Enable the logger in the digi.xbee.devices module with INFO level.
dev_logger = enable_logger(digi.xbee.devices.__name__, logging.INFO)

# This is a valid method to do the same.
dev_logger = enable_logger("digi.xbee.devices", logging.INFO)

# Enable the logger in the digi.xbee.devices module with the default level
# (DEBUG).
dev_logger = enable_logger("digi.xbee.devices")

# This is a valid method to do the same.
dev_logger = enable_logger("digi.xbee.devices", logging.DEBUG)
```

(...)

(continued from previous page)
2.5.10 XBee Python samples

The XBee Python Library includes several samples to demonstrate how to do the following:

- Communicate with your modules
- Configure your modules
- Read the IO lines
- Update device’s firmware
- Work with device’s file system
- Apply XBee profiles
- Perform other common operations

All of the sample applications are contained in the examples folder, organized by category. Every sample includes the source code and a readme.txt file to clarify the purpose and the required setup to launch the application.

Examples are split by categories:

- Configuration samples
- Network samples
- Communication samples
- IO samples
- Firmware samples
- File system samples
- Profile samples

2.5.10.1 Configuration samples

Manage common parameters

This sample application shows how to get and set common parameters of the XBee device. Common parameters are split in cached and non-cached parameters. For that reason, the application refreshes the cached parameters before reading and displaying them. The application then configures, reads, and displays the value of non-cached parameters.

The application uses the specific setters and getters provided by the XBee device object to configure and read the different parameters.

You can locate the example in the following path: examples/configuration/ManageCommonParametersSample

Note: For more information about how to manage common parameters, see Read and set common parameters.
Set and get parameters

This sample application shows how to set and get parameters of a local or remote XBee device. Use this method when you need to set or get the value of a parameter that does not have its own getter and setter within the XBee device object.

The application sets the value of four parameters with different value types:

- String
- Byte
- Array
- Integer

The application then reads the parameters from the device to verify that the read values are the same as the values that were set.

You can locate the example in the following path: examples/configuration/SetAndGetParametersSample

Note: For more information about how to get and set other parameters, see Read, set and execute other parameters.

Reset module

This sample application shows how to perform a software reset on the local XBee module.

You can locate the example in the following path: examples/configuration/ResetModuleSample

Note: For more information about how to reset a module, see Reset the device.

Recover XBee serial connection

This sample application shows how to recover the serial settings of a local XBee.

You can locate the example at the following path: examples/configuration/RecoverSerialConnection

Note: For more information about this, see Open the XBee device connection.

Connect to access point (Wi-Fi)

This sample application shows how to configure a Wi-Fi module to connect to a specific access point and read its addressing settings.

You can locate the example at the following path: examples/configuration/ConnectToAccessPoint

Note: For more information about connecting to an access point, see Configure Wi-Fi settings.
2.5.10.2 Network samples

Discover devices

This sample application demonstrates how to obtain the XBee network object from a local XBee device and discover the remote XBee devices that compose the network. The example adds a discovery listener, so the callbacks provided by the listener object receive the events.

The remote XBee devices are printed out as soon as they are found during discovery.

You can locate the example in the following path: examples/network/DiscoverDevicesSample

Note: For more information about how to perform a network discovery, see Discover the network.

Network modifications sample

This sample application demonstrates how to listen to network modification events. The example adds a modifications network callback, so modifications events are received and printed out.

A network is modified when:

- a new node is added by discovering, manually, or because data is received from it
- an existing node is removed from the network
- an existing node is updated with new information
- it is fully cleared

You can locate the example in the following path: examples/network/NetworkModificationsSample

Note: For more information about how to listen to network modifications, see Listen to network modification events.

2.5.10.3 Communication samples

Send data

This sample application shows how to send data from the XBee device to another remote device on the same network using the XBee Python Library. In this example, the application sends data using a reliable transmission method. The application blocks during the transmission request, but you are notified if there is any error during the process.

The application sends data to a remote XBee device on the network with a specific node identifier (name).

You can locate the example in the following path: examples/communication/SendDateSample

Note: For more information about how to send data, see Send data.
Send data asynchronously

This sample application shows how to send data asynchronously from the XBee device to another remote device on the same network using the XBee Python Library. Transmitting data asynchronously means the execution is not blocked during the transmit request, but you cannot determine if the data was sent successfully.

The application sends data asynchronously to a remote XBee device on the network with a specific node identifier (name).

You can locate the example in the following path: examples/communication/SendDataAsyncSample

Note: For more information about how to send data, see Send data.

Send broadcast data

This sample application shows how to send data from the local XBee device to all remote devices on the same network (broadcast) using the XBee Python Library. The application blocks during the transmission request, but you are notified if there is any error during the process.

You can locate the example in the following path: examples/communication/SendBroadcastDataSample

Note: For more information about how to send broadcast data, see Send data to all devices of the network.

Send explicit data

This sample application shows how to send data in application layer (explicit) format to a remote Zigbee device on the same network as the local one using the XBee Python Library. In this example, the XBee module sends explicit data using a reliable transmission method. The application blocks during the transmission request, but you are notified if there is any error during the process.

You can locate the example in the following path: examples/communication/explicit/SendExplicitDataSample

Note: For more information about how to send explicit data, see Send explicit data.

Send explicit data asynchronously

This sample application shows how to send data in application layer (explicit) format asynchronously to a remote Zigbee device on the same network as the local one using the XBee Python Library. Transmitting data asynchronously means the execution is not blocked during the transmit request, but you cannot determine if the data was sent successfully.

You can locate the example in the following path: examples/communication/explicit/SendExplicitDataAsyncSample

Note: For more information about how to send explicit data, see Send explicit data.
Send broadcast explicit data

This sample application shows how to send data in application layer (explicit) format to all remote devices on the same network (broadcast) as the local one using the XBee Python Library. The application blocks during the transmission request, but you are notified if there is any error during the process.

You can locate the example in the following path: examples/communication/explicit/SendBroadcastExplicitDataSample

Note: For more information about how to send broadcast explicit data, see Send explicit data to all devices in the network.

Send IP data (IP devices)

This sample application shows how to send IP data to another device specified by its IP address and port number.

You can find the example at the following path: examples/communication/ip/SendIPDataSample

Note: For more information about how to send IP data, see Send IP data.

Send SMS (cellular devices)

This sample application shows how to send an SMS to a phone or cellular device.

You can find the example at the following path: examples/communication/cellular/SendSMSSample

Note: For more information about how to send SMS messages, see Send SMS messages.

Send UDP data (IP devices)

This sample application shows how to send UDP data to another device specified by its IP address and port number.

You can find the example at the following path: examples/communication/ip/SendUDPDataSample

Note: For more information about how to send IP data, see Send IP data.

Send Bluetooth Data

This sample application shows how to send data to the XBee Bluetooth Low Energy interface.

You can find the example at the following path: examples/communication/bluetooth/SendBluetoothDataSample

Note: For more information about sending Bluetooth data, see Send Bluetooth data.
Send MicroPython Data

This sample application shows how to send data to the XBee MicroPython interface.

You can find the example at the following path: examples/communication/micropython/SendMicroPythonDataSample

Note: For more information about sending MicroPython data, see Send MicroPython data.

Send User Data Relay

This sample application shows how to send data to other XBee interface.

You can find the example at the following path: examples/communication/relay/SendUserDataRelaySample

Note: For more information about sending User Data Relay messages, see Send Bluetooth data or Send MicroPython data.

Receive data

This sample application shows how data packets are received from another XBee device on the same network.

The application prints the received data to the standard output in ASCII and hexadecimal formats after the sender address.

You can locate the example in the following path: examples/communication/ReceiveDataSample

Note: For more information about how to receive data using a callback, see Data reception callback.

Receive data polling

This sample application shows how data packets are received from another XBee device on the same network using a polling mechanism.

The application prints the data that was received to the standard output in ASCII and hexadecimal formats after the sender address.

You can locate the example in the following path: examples/communication/ReceiveDataPollingSample

Note: For more information about how to receive data using a polling mechanism, see Polling for data.

Receive explicit data

This sample application shows how a Zigbee device receives data in application layer (explicit) format using a callback executed every time new data is received. Before receiving data in explicit format, the API output mode of the Zigbee device is configured in explicit mode.

You can locate the example in the following path: examples/communication/explicit/ReceiveExplicitDataSample
Note: For more information about how to receive explicit data using a callback, see *Explicit data reception callback*.

### Receive explicit data polling

This sample application shows how a Zigbee device receives data in application layer (explicit) format using a polling mechanism. Before receiving data in explicit format, the API output mode of the Zigbee device is configured in explicit mode.

You can locate the example in the following path: `examples/communication/explicit/ReceiveExplicitDataPollingSample`

Note: For more information about how to receive explicit data using a polling mechanism, see *Polling for explicit data*.

### Receive IP data (IP devices)

This sample application shows how an IP device receives IP data using a callback executed every time it receives new IP data.

You can find the example at the following path: `examples/communication/ip/ReceiveIPDataSample`

Note: For more information about how to receive IP data using a polling mechanism, see *Receive IP data*.

### Receive SMS (cellular devices)

This sample application shows how to receive SMS messages configuring a callback executed when new SMS is received.

You can find the example at the following path: `examples/communication/cellular/ReceiveSMSSample`

Note: For more information about how to receive SMS messages, see *Receive SMS messages*.

### Receive Bluetooth data

This sample application shows how to receive data from the XBee Bluetooth Low Energy interface.

You can find the example at the following path: `examples/communication/bluetooth/ReceiveBluetoothDataSample`

Note: For more information about receiving Bluetooth data, see *Receive Bluetooth data*.

### Receive Bluetooth file

This sample application shows how to receive a file from the XBee Bluetooth Low Energy interface.
You can find the example at the following path: `examples/communication/bluetooth/ReceiveBluetoothFileSample`

**Note:** For more information about receiving Bluetooth data, see *Receive Bluetooth data.*

---

**Receive MicroPython data**

This sample application shows how to receive data from the XBee MicroPython interface.

You can find the example at the following path: `examples/communication/micropython/ReceiveMicroPythonDataSample`

**Note:** For more information about receiving MicroPython data, see *Receive MicroPython data.*

---

**Receive User Data Relay**

This sample application shows how to receive data from other XBee interface.

You can find the example at the following path: `examples/communication/relay/ReceiveUserDataRelaySample`

**Note:** For more information about receiving User Data Relay messages, see *Receive Bluetooth data* or *Receive MicroPython data.*

---

**Receive modem status**

This sample application shows how modem status packets (events related to the device and the network) are handled using the API.

The application prints the modem status events to the standard output when received.

You can locate the example in the following path: `examples/communication/ReceiveModemStatusSample`

**Note:** For more information about how to receive modem status events, see *Receive modem status events.*

---

**Connect to echo server (IP devices)**

This sample application shows how IP devices can connect to an echo server, send data to it and reads the echoed data.

You can find the example at the following path: `examples/communication/ip/ConnectToEchoServerSample`

**Note:** For more information about how to send and receive IP data, see *Send IP data* and *Receive IP data.*
Create a TCP client socket (cellular devices)

This sample application shows how to create a TCP client socket to send HTTP requests.
You can find the example at the following path: examples/communication/socket/SocketTCPClientSample

Note: For more information about how to use the XBee socket API, see Communicate using XBee sockets.

Create a TCP server socket (cellular devices)

This sample application shows how to create a TCP server socket to receive data from incoming sockets.
You can find the example at the following path: examples/communication/socket/SocketTCPServerSample

Note: For more information about how to use the XBee socket API, see Communicate using XBee sockets.

Create a UDP server/client socket (cellular devices)

This sample application shows how to create a UDP socket to deliver messages to a server and listen for data coming from multiple peers.
You can find the example at the following path: examples/communication/socket/SocketUDPServerClientSample

Note: For more information about how to use the XBee socket API, see Communicate using XBee sockets.

2.5.10.4 IO samples

Local DIO

This sample application shows how to set and read XBee digital lines of the device attached to the serial/USB port of your PC.
The application configures two IO lines of the XBee device: one as a digital input (button) and the other as a digital output (LED). The application reads the status of the input line periodically and updates the output to follow the input.
The LED lights up while you press the button.
You can locate the example in the following path: examples/io/LocalDIOSample

Note: For more information about how to set and read digital lines, see Digital Input/Output.

Local ADC

This sample application shows how to read XBee analog inputs of the device attached to the serial/USB port of your PC.
The application configures an IO line of the XBee device as ADC. It periodically reads its value and prints it in the output console.

You can locate the example in the following path: `examples/io/LocalADCSample`

**Note:** For more information about how to read analog lines, see `ADC`.

### Remote DIO

This sample application shows how to set and read XBee digital lines of remote devices.

The application configures two IO lines of the XBee devices: one in the remote device as a digital input (button) and the other in the local device as a digital output (LED). The application reads the status of the input line periodically and updates the output to follow the input.

The LED lights up while you press the button.

You can locate the example in the following path: `examples/io/RemoteDIOSample`

**Note:** For more information about how to set and read digital lines, see `Digital Input/Output`.

### Remote ADC

This sample application shows how to read XBee analog inputs of remote XBee devices.

The application configures an IO line of the remote XBee device as ADC. It periodically reads its value and prints it in the output console.

You can locate the example in the following path: `examples/io/RemoteADCSample`

**Note:** For more information about how to read analog lines, see `ADC`.

### IO sampling

This sample application shows how to configure a remote device to send automatic IO samples and how to read them from the local module.

The application configures two IO lines of the remote XBee device: one as digital input (button) and the other as ADC, and enables periodic sampling and change detection. The device sends a sample every five seconds containing the values of the two monitored lines. The device sends another sample every time the button is pressed or released, which only contains the value of this digital line.

The application registers a listener in the local device to receive and handle all IO samples sent by the remote XBee module.

You can locate the example in the following path: `examples/io/IOSamplingSample`

**Note:** For more information about how to read IO samples, see `Read IO samples`. 
2.5.10.5 Firmware samples

Update local firmware

This sample Python application shows how to update the firmware of a local XBee device.
The application provides the required hardware files to the update method as well as a callback function to be notified of progress.
You can locate the example in the following path: examples/firmware/LocalFirmwareUpdateSample

Update remote firmware

This sample Python application shows how to update the firmware of a remote XBee device.
The application provides the required hardware files to the update method as well as a callback function to be notified of progress.
You can locate the example in the following path: examples/firmware/RemotelFirmwareUpdateSample

2.5.10.6 File system samples

Format file system

This sample Python application shows how to format the filesystem of a local XBee device and retrieve usage information.
The application uses the LocalXBeeFileSystemManager to access the device filesystem and execute the required actions.
You can locate the example in the following path: examples/filesystem/FormatFilesystemSample

List directory contents

This sample Python application shows how to list the contents of an XBee device filesystem directory.
The application uses the LocalXBeeFileSystemManager to access the device filesystem and executes the required actions.
You can locate the example in the following path: examples/filesystem/ListDirectorySample

Upload/download file

This sample Python application shows how to upload and download a file from a local XBee device filesystem.
The application uses the LocalXBeeFileSystemManager to access the device filesystem and provides the local file and the necessary paths to the upload/download methods as well as callback functions to be notified of progress.
You can locate the example in the following path: examples/filesystem/UploadDownloadFileSample
2.5.10.7 Profile samples

Apply local profile

This sample Python application shows how to apply an existing XBee profile to a XBee device. The application provides the profile file to the update method as well as a callback function to be notified of progress. You can locate the example in the following path: examples/profile/ApplyXBeeProfileSample

Apply remote profile

This sample Python application shows how to apply an existing XBee profile to a remote XBee device. The application provides the profile file to the update method as well as a callback function to be notified of progress. You can locate the example in the following path: examples/profile/ApplyXBeeProfileRemoteSample

Read profile

This sample Python application shows how to read an existing XBee profile and extract its properties. The application creates an XBee profile object from an existing XBee profile file and prints all the accessible settings and properties. You can locate the example in the following path: examples/profile/ReadXBeeProfileSample

2.5.11 Frequently Asked Questions (FAQs)

The FAQ section contains answers to general questions related to the XBee Python Library.

2.5.11.1 What is XCTU and how do I download it?

XCTU is a free multi-platform application designed to enable developers to interact with Digi RF modules through a simple-to-use graphical interface. You can download it at www.digi.com/xctu.

2.5.11.2 How do I find the serial port and baud rate of my module?

Open the XCTU application, and click the Discover radio modules connected to your machine button. Select all ports to be scanned, click Next and then Finish. Once the discovery process has finished, a new window notifies you how many devices have been found and their details. The serial port and the baud rate are shown in the Port label.
2.5.11.3 Can I use the XBee Python Library with modules in AT operating mode?

No, the XBee Python Library only supports API and API Escaped operating modes.

2.5.11.4 I get the Python error ImportError: No module named 'serial'

This error means that Python cannot find the `serial` module, which is used by the library for the serial communication with the XBee devices.

Note: Note In UNIX systems, the complete name of the serial port contains the `/dev/` prefix.
You can install PySerial running this command in your terminal application:

```
$ pip install pyserial
```

For further information about the installation of PySerial, refer to the PySerial installation guide.

### 2.5.11.5 I get the Python error ImportError: No module named 'srp'

This error means that Python cannot find the `srp` module, which is used by the library to authenticate with XBee devices over Bluetooth Low Energy.

You can install SRP running this command in your terminal application:

```
$ pip install srp
```

### 2.5.12 API reference

Following is API reference material on major parts of XBee Python library.

#### 2.5.12.1 digi package

**Subpackages**

- `digi.xbee` package

**Subpackages**

- `digi.xbee.models` package

**Submodules**

- `digi.xbee.models.accesspoint` module

```python
class digi.xbee.models.accesspoint.AccessPoint(ssid, encryption_type, channel=0, signal_quality=0)
```

-Bases: object

This class represents an Access Point for the Wi-Fi protocol. It contains SSID, the encryption type and the link quality between the Wi-Fi module and the access point.

This class is used within the library to list the access points and connect to a specific one in the Wi-Fi protocol.

See also:

- `WiFiEncryptionType`

Class constructor. Instantiates a new `AccessPoint` object with the provided parameters.

**Parameters**

- `ssid (String)` – the SSID of the access point.
• **encryption_type** (*WiFiEncryptionType*) – the encryption type configured in the access point.

• **channel** (*Integer, optional*) – operating channel of the access point. Optional.

• **signal_quality** (*Integer, optional*) – signal quality with the access point in %. Optional.

Raises

• **ValueError** – if length of `ssid` is 0.

• **ValueError** – if `channel` is less than 0.

• **ValueError** – if `signal_quality` is less than 0 or greater than 100.

See also:

*WiFiEncryptionType*

```python
ssid
String. SSID of the access point.

encryption_type
    *WiFiEncryptionType*. Encryption type of the access point.

channel
String. Channel of the access point.

signal_quality
String. The signal quality with the access point in %.
```

class *digi.xbee.models.accesspoint.WiFiEncryptionType*(code, description)

Bases: *enum.Enum*

Enumerates the different Wi-Fi encryption types.

Values:

- *WiFiEncryptionType.NONE* = (0, ‘No security’)
- *WiFiEncryptionType.WPA* = (1, ‘WPA (TKIP) security’)
- *WiFiEncryptionType.WPA2* = (2, ‘WPA2 (AES) security’)
- *WiFiEncryptionType.WEP* = (3, ‘WEP security’)

```python
code
Integer. The Wi-Fi encryption type code.

description
String. The Wi-Fi encryption type description.
```

digi.xbee.models.atcomm module

class *digi.xbee.models.atcomm.ATStringCommand*(command, description)

Bases: *enum.Enum*
This class represents basic AT commands.

Inherited properties:

- **name** (String): name (ID) of this ATStringCommand.
- **value** (String): value of this ATStringCommand.

Values:

- `ATStringCommand.AC = ('AC', 'Apply changes')`
- `ATStringCommand.AG = ('AG', 'Aggregator support')`
- `ATStringCommand.AI = ('AI', 'Association indication')`
- `ATStringCommand.AO = ('AO', 'API options')`
- `ATStringCommand.AP = ('AP', 'API enable')`
- `ATStringCommand.AR = ('AR', 'Many-to-one route broadcast time')`
- `ATStringCommand.AS = ('AS', 'Active scan')`
- `ATStringCommand.BD = ('BD', 'UART baudrate')`
- `ATStringCommand.BL = ('BL', 'Bluetooth address')`
- `ATStringCommand.BT = ('BT', 'Bluetooth enable')`
- `ATStringCommand.C0 = ('C0', 'Source port')`
- `ATStringCommand.C8 = ('C8', 'Compatibility mode')`
- `ATStringCommand.CC = ('CC', 'Command sequence character')`
- `ATStringCommand.CE = ('CE', 'Device role')`
- `ATStringCommand.CN = ('CN', 'Exit command mode')`
- `ATStringCommand.DA = ('DA', 'Force Disassociation')`
- `ATStringCommand.DH = ('DH', 'Destination address high')`
- `ATStringCommand.DL = ('DL', 'Destination address low')`
- `ATStringCommand.D7 = ('D7', 'CTS configuration')`
- `ATStringCommand.EE = ('EE', 'Encryption enable')`
- `ATStringCommand.FN = ('FN', 'Find neighbors')`
- `ATStringCommand.FR = ('FR', 'Software reset')`
- `ATStringCommand.FS = ('FS', 'File system')`
- `ATStringCommand.GW = ('GW', 'Gateway address')`
- `ATStringCommand.GT = ('GT', 'Guard times')`
- `ATStringCommand.HV = ('HV', 'Hardware version')`
- `ATStringCommand.IC = ('IC', 'Digital change detection')`
- `ATStringCommand.ID = ('ID', 'Network PAN ID/Network ID/SSID')`
- `ATStringCommand.IR = ('IR', 'I/O sample rate')`
- `ATStringCommand.IS = ('IS', 'Force sample')`
- `ATStringCommand.KY = ('KY', 'Link/Encryption key')`
- `ATStringCommand.MA = ('MA', 'IP addressing mode')`
- `ATStringCommand.MK = ('MK', 'IP address mask')`
- `ATStringCommand.MY = ('MY', '16-bit address/IP address')`
- `ATStringCommand.NB = ('NB', 'Parity')`
- `ATStringCommand.NI = ('NI', 'Node identifier')`
- `ATStringCommand.ND = ('ND', 'Node discover')`
- `ATStringCommand.NK = ('NK', 'Trust Center network key')`
- `ATStringCommand.NO = ('NO', 'Node discover options')`
ATStringCommand.NR = ('NR', 'Network reset')
ATStringCommand.NS = ('NS', 'DNS address')
ATStringCommand.NT = ('NT', 'Node discover back-off')
ATStringCommand.N_QUESTION = ('N?', 'Network discovery timeout')
ATStringCommand.OP = ('OP', 'Operating extended PAN ID')
ATStringCommand.PK = ('PK', 'Passphrase')
ATStringCommand.PL = ('PL', 'TX power level')
ATStringCommand.RE = ('RE', 'Restore defaults')
ATStringCommand.RR = ('RR', 'XBee retries')
ATStringCommand.R_QUESTION = ('R?', 'Region lock')
ATStringCommand.SB = ('SB', 'Stop bits')
ATStringCommand.SH = ('SH', 'Serial number high')
ATStringCommand.SI = ('SI', 'Socket info')
ATStringCommand.SL = ('SL', 'Serial number low')
ATStringCommand.SM = ('SM', 'Sleep mode')
ATStringCommand.SS = ('SS', 'Sleep status')
ATStringCommand.VH = ('VH', 'Bootloader version')
ATStringCommand.VR = ('VR', 'Firmware version')
ATStringCommand.WR = ('WR', 'Write')
ATStringCommand.DOLLAR_S = ('$S', 'SRP salt')
ATStringCommand.DOLLAR_V = ('$V', 'SRP salt verifier')
ATStringCommand.DOLLAR_W = ('$W', 'SRP salt verifier')
ATStringCommand.DOLLAR_X = ('$X', 'SRP salt verifier')
ATStringCommand.DOLLAR_Y = ('$Y', 'SRP salt verifier')
ATStringCommand.PERCENT_C = ('%C', 'Hardware/software compatibility')
ATStringCommand.PERCENT_P = ('%P', 'Invoke bootloader')

class digi.xbee.models.atcomm.SpecialByte(code)

    Bases: enum.Enum

    Enumerates all the special bytes of the XBee protocol that must be escaped when working on API 2 mode.

    Inherited properties:
    - name (String): name (ID) of this SpecialByte.
    - value (String): the value of this SpecialByte.

    Values:
    - SpecialByte.ESCAPE_BYTE = 125
    - SpecialByte.HEADER_BYTE = 126
    - SpecialByte.XON_BYTE = 17
code
  Integer. The special byte code.

class digi.xbee.models.atcomm.ATCommand(command, parameter=None)
  Bases: object
  This class represents an AT command used to read or set different properties of the XBee device.
  AT commands can be sent directly to the connected device or to remote devices and may have parameters.
  After executing an AT Command, an AT Response is received from the device.
  Class constructor. Instantiates a new ATCommand object with the provided parameters.

  Parameters
  • command(String) – AT Command, must have length 2.
  • parameter(String or Bytearray, optional) – The AT parameter value. Defaults to None. Optional.

  Raises ValueError – if command length is not 2.

get_parameter_string()
  Returns this ATCommand parameter as a String.

    Returns this ATCommand parameter. None if there is no parameter.

    Return type String

command
  String. The AT command

parameter
  Bytearray. The AT command parameter

class digi.xbee.models.atcomm.ATCommandResponse(command, response=None, status=<ATCommandStatus.OK: (0, 'Status OK')>)
  Bases: object
  This class represents the response of an AT Command sent by the connected XBee device or by a remote device
  after executing an AT Command.
  Class constructor.

  Parameters
  • command(ATCommand) – The AT command that generated the response.
  • response(bytearray, optional) – The command response. Default to None.
  • status(ATCommandStatus, optional) – The AT command status. Default to ATCommandStatus.OK

command
  String. The AT command.

response
  Bytearray. The AT command response data.
status

ATCommandStatus. The AT command response status.

digi.xbee.models.hw module

class digi.xbee.models.hw.HardwareVersion (code, description)

    Bases: enum.Enum

    This class lists all hardware versions.

    Inherited properties:
    
    name (String): The name of this HardwareVersion.
    value (Integer): The ID of this HardwareVersion.

    Values:
    
    HardwareVersion.X09_009 = (1, ‘X09-009’)
    HardwareVersion.X09_019 = (2, ‘X09-019’)
    HardwareVersion.XH9_009 = (3, ‘XH9-009’)
    HardwareVersion.XH9_019 = (4, ‘XH9-019’)
    HardwareVersion.X24_009 = (5, ‘X24-009’)
    HardwareVersion.X24_019 = (6, ‘X24-019’)
    HardwareVersion.X09_001 = (7, ‘X09-001’)
    HardwareVersion.XH9_001 = (8, ‘XH9-001’)
    HardwareVersion.X08_004 = (9, ‘X08-004’)
    HardwareVersion.XC09_009 = (10, ‘XC09-009’)
    HardwareVersion.XC09_038 = (11, ‘XC09-038’)
    HardwareVersion.X24_038 = (12, ‘X24-038’)
    HardwareVersion.X09_009_TX = (13, ‘X09-009-TX’)
    HardwareVersion.X09_019_TX = (14, ‘X09-019-TX’)
    HardwareVersion.XH9_009_TX = (15, ‘XH9-009-TX’)
    HardwareVersion.XH9_019_TX = (16, ‘XH9-019-TX’)
    HardwareVersion.X09_001_TX = (17, ‘X09-001-TX’)
    HardwareVersion.XH9_001_TX = (18, ‘XH9-001-TX’)
    HardwareVersion.XT09B_XXX = (19, ‘XT09B-xxx (Attenuator version)’)
    HardwareVersion.XT09_XXX = (20, ‘XT09-xxx’)
    HardwareVersion.XC08_009 = (21, ‘XC08-009’)
    HardwareVersion.XC08_038 = (22, ‘XC08-038’)
    HardwareVersion.XB24_AXX_XX = (23, ‘XB24-Axx-xx’)
    HardwareVersion.XBP24_AXX_XX = (24, ‘XBP24-Axx-xx’)
    HardwareVersion.XB24_BXIX_XXX = (25, ‘XB24-BxIx-xxx and XB24-Z7xx-xxx’)
    HardwareVersion.XBP24_BXIX_XXX = (26, ‘XBP24-BxIx-xxx and XBP24-Z7xx-xxx’)
    HardwareVersion.XBP09_DXIX_XXX = (27, ‘XBP09-DxIx-xxx Digi Mesh’)
    HardwareVersion.XBP09_XCXX_XXX = (28, ‘XBP09-XCxx-xxx: S3 XSC Compatibility’)
    HardwareVersion.XBP08_DXXX_XXX = (29, ‘XBP08-Dxx-xxx 868MHz’)
    HardwareVersion.XBP24B = (30, ‘XBP24B: Low cost ZB PRO and PLUS S2B’)
    HardwareVersion.XB24_WF = (31, ‘XB24-WF: XBee 802.11 (Redpine module)’)

2.5. API reference
HardwareVersion.AMBER_MBUS = (32, ‘???????: M-Bus module made by Amber’)
HardwareVersion.XBP24C = (33, ‘XBP24C: XBee PRO SMT Ember 357 S2C PRO’)
HardwareVersion.XB24C = (34, ‘XB24C: XBee SMT Ember 357 S2C’)
HardwareVersion.XSC_GEN3 = (35, ‘XSC_GEN3: XBP9 XSC 24 dBm’)
HardwareVersion.SRD_868_GEN3 = (36, ‘SDR_868_GEN3: XB8 12 dBm’)
HardwareVersion.ABANDONATED = (37, ‘Abandoned’)
HardwareVersion.SMT_900LP = (38, “900LP (SMT): 900LP on ‘S8 HW’”)
HardwareVersion.WIFI_ATHEROS = (39, ‘WiFi Atheros (TH-DIP) XB2S-WF’)
HardwareVersion.SMT_WIFI_ATHEROS = (40, ‘WiFi Atheros (SMT) XB2B-WF’)
HardwareVersion.SMT_475LP = (41, ‘475LP (SMT): Beta 475MHz’)
HardwareVersion.XBEE_CELL_TH = (42, ‘XBee-Cell (TH): XBee Cellular’)
HardwareVersion.XLR_MODULE = (43, ‘XLR Module’)
HardwareVersion.XB900HP_NZ = (44, ‘XB900HP (New Zealand): XB9 NZ HW/SW’)
HardwareVersion.XBP24C_TH_DIP = (45, ‘XBP24C (TH-DIP): XBee PRO DIP’)
HardwareVersion.XB24C_TH_DIP = (46, ‘XB24C (TH-DIP): XBee DIP’)
HardwareVersion.XLR_BASEBOARD = (47, ‘XLR Baseboard’)
HardwareVersion.XBP24C_S2C_SMT = (48, ‘XBee PRO SMT’)
HardwareVersion.SX_PRO = (49, ‘SX Pro’)
HardwareVersion.S2D_SMT_PRO = (50, ‘XBP24D: S2D SMT PRO’)
HardwareVersion.S2D_SMT_REG = (51, ‘XB24D: S2D SMT Reg’)
HardwareVersion.S2D_TH_PRO = (52, ‘XBP24D: S2D TH PRO’)
HardwareVersion.S2D_TH_REG = (53, ‘XB24D: S2D TH Reg’)
HardwareVersion.SX = (62, ‘SX’)
HardwareVersion.XTR = (63, ‘XTR’)
HardwareVersion.CELLULAR_CAT1_LTE_VERIZON = (64, ‘XBee Cellular Cat 1 LTE Verizon’)
HardwareVersion.XBEE3 = (65, ‘XBEE3’)
HardwareVersion.XBEE3_SMT = (66, ‘XBEE3 SMT’)
HardwareVersion.XUBE3_TH = (67, ‘XBEE3 TH’)
HardwareVersion.CELLULAR_3G = (68, ‘XBee Cellular 3G’)
HardwareVersion.XB8X = (69, ‘XB8X’)
HardwareVersion.CELLULAR_LTE_VERIZON = (70, ‘XBee Cellular LTE-M Verizon’)
HardwareVersion.CELLULAR_LTE_ATT = (71, ‘XBee Cellular LTE-M AT&T’)
HardwareVersion.CELLULAR_NBLOT_EUROPE = (72, ‘XBee Cellular NBloT Europe’)
HardwareVersion.CELLULAR_3_CAT1_LTE_ATT = (73, ‘XBee Cellular 3 Cat 1 LTE AT&T’)
HardwareVersion.CELLULAR_3_LTE_M_VERIZON = (74, ‘XBee Cellular 3 LTE-M Verizon’)
HardwareVersion.CELLULAR_3_LTE_M_ATT = (75, ‘XBee Cellular 3 LTE-M AT&T’)

code
Integer. The hardware version code.

description
String. The hardware version description.
digi.xbee.models.mode module

class digi.xbee.models.mode.OperatingMode (code, description)
    Bases: enum.Enum

    This class represents all operating modes available.

    Inherited properties:
    name (String): the name (id) of this OperatingMode.
    value (String): the value of this OperatingMode.

    Values:
    OperatingMode.AT_MODE = (0, ‘AT mode’)
    OperatingMode.API_MODE = (1, ‘API mode’)
    OperatingMode.ESCAPED_API_MODE = (2, ‘API mode with escaped characters’)
    OperatingMode.MICROPYTHON_MODE = (4, ‘MicroPython REPL’)
    OperatingMode.BYPASS_MODE = (5, ‘Bypass mode’)
    OperatingMode.UNKNOWN = (99, ‘Unknown’)

code
    Integer. The operating mode code.

description
    The operating mode description.
      Type  String

class digi.xbee.models.mode.APIOutputMode (code, description)
    Bases: enum.Enum

    Enumerates the different API output modes. The API output mode establishes the way data will
    be output through the serial interface of an XBee device.

    Inherited properties:
    name (String): the name (id) of this OperatingMode.
    value (String): the value of this OperatingMode.

    Values:
    APIOutputMode.NATIVE = (0, ‘Native’)
    APIOutputMode.EXPLICIT = (1, ‘Explicit’)
    APIOutputMode.EXPLICIT_ZDO_PASSTHRU = (3, ‘Explicit with ZDO Passthru’)

code
    Integer. The API output mode code.
description
The API output mode description.

Type String
class digi.xbee.models.mode.APIOutputModeBit(code, description)
Bases: enum.Enum

Enumerates the different API output mode bit options. The API output mode establishes the way
data will be output through the serial interface of an XBee.

Inherited properties:
name (String): the name (id) of this APIOutputModeBit.
value (String): the value of this APIOutputModeBit.

Values:
APIOutputModeBit.EXPLICIT = (1, ‘Output in Native/Explicit API format’)
APIOutputModeBit.UNSUPPORTED_ZDO_PASSTHRU = (2, ‘Unsupported ZDO request pass-through’)
APIOutputModeBit.SUPPORTED_ZDO_PASSTHRU = (4, ‘Supported ZDO request pass-through’)
APIOutputModeBit.BINDING_PASSTHRU = (8, ‘Binding request pass-through’)

code
Integer. The API output mode bit code.
description
The API output mode bit description.

class digi.xbee.models.mode.IPAddressingMode(code, description)
Bases: enum.Enum

Enumerates the different IP addressing modes.

Values:
IPAddressingMode.DHCP = (0, ‘DHCP’)
IPAddressingMode.STATIC = (1, ‘Static’)

code
Integer. The IP addressing mode code.
description
String. The IP addressing mode description.
class digi.xbee.models.mode.NeighborDiscoveryMode(code, description)
Bases: enum.Enum

Enumerates the different neighbor discovery modes. This mode establishes the way the network discovery
process is performed.
Inherited properties:

- **name** (String): the name (id) of this OperatingMode.
- **value** (String): the value of this OperatingMode.

**CASCADE = (0, 'Cascade')**

The discovery of a node neighbors is requested once the previous request finishes. This means that just one discovery process is running at the same time.

This mode is recommended for large networks, it might be a slower method but it generates less traffic than ‘Flood’.

**FLOOD = (1, 'Flood')**

The discovery of a node neighbors is requested when the node is found in the network. This means that several discovery processes might be running at the same time.

**code**

Returs the code of the NeighborDiscoveryMode element.

- **Returns** the code of the NeighborDiscoveryMode element.
- **Return type** String

**description**

Returns the description of the NeighborDiscoveryMode element.

- **Returns** the description of the NeighborDiscoveryMode element.
- **Return type** String

---

digi.xbee.models.address module

class digi.xbee.models.address.XBee16BitAddress

**Bases:** object

This class represent a 16-bit network address.

This address is only applicable for:

1. 802.15.4
2. ZigBee
3. ZNet 2.5
4. XTend (Legacy)

DigiMesh and Point-to-multipoint does not support 16-bit addressing.

Each device has its own 16-bit address which is unique in the network. It is automatically assigned when the radio joins the network for ZigBee and Znet 2.5, and manually configured in 802.15.4 radios.

**Attributes:**

- **COORDINATOR_ADDRESS** (XBee16BitAddress): 16-bit address reserved for the coordinator.
- **BROADCAST_ADDRESS** (XBee16BitAddress): 16-bit broadcast address.
- **UNKNOWN_ADDRESS** (XBee16BitAddress): 16-bit unknown address.
- **PATTERN** (String): Pattern for the 16-bit address string: \([0-9a-fA-F]{1,4}\)
Class constructor. Instantiates a new `XBee16BitAddress` object with the provided parameters.

**Parameters**  
`address (Bytearray)` – address as byte array. Must be 1-2 digits.

**Raises**  
- `TypeError` – if `address` is `None`.  
- `ValueError` – if `address` is `None` or has less than 1 byte or more than 2.

**PATTERN** = `'^(0[xX])?[0-9a-fA-F]{1,4}$'`  
16-bit address string pattern.

**COORDINATOR_ADDRESS** = `<digi.xbee.models.address.XBee16BitAddress object> (0000)`.  
Type 16-bit address reserved for the coordinator (value 0000).

**BROADCAST_ADDRESS** = `<digi.xbee.models.address.XBee16BitAddress object> (FFFF)`.  
Type 16-bit broadcast address (value FFFF).

**UNKNOWN_ADDRESS** = `<digi.xbee.models.address.XBee16BitAddress object> (FFFE)`.  
Type 16-bit unknown address (value FFFE).

**classmethod from_hex_string**(`address`)  
Class constructor. Instantiates a new `XBee16BitAddress` object from the provided hex string.

**Parameters**  
`address (String)` – String containing the address. Must be made by hex. digits without blanks. Minimum 1 character, maximum 4 (16-bit).

**Raises**  
- `ValueError` – if `address` has less than 1 character.  
- `ValueError` – if `address` contains non-hexadecimal characters.

**classmethod from_bytes**(`hsb, lsb`)  
Class constructor. Instantiates a new `XBee16BitAddress` object from the provided high significant byte and low significant byte.

**Parameters**  
- `hsb (Integer)` – high significant byte of the address.  
- `lsb (Integer)` – low significant byte of the address.

**Raises**  
- `ValueError` – if `lsb` is less than 0 or greater than 255.  
- `ValueError` – if `hsb` is less than 0 or greater than 255.

**classmethod is_valid**(`address`)  
Checks if the provided hex string is a valid 16-bit address.

**Parameters**  
`address (String or Bytearray)` – String: String containing the address. Must be made by hex. digits without blanks. Minimum 1 character, maximum 4 (16-bit). Bytearray: Address as byte array. Must be 1-2 digits.

**Returns** `True` for a valid 16-bit address, `False` otherwise.

**Return type**  
Boolean
**get_hsb()**
Returns the high part of the bytearray (component 0).

Returns: high part of the bytearray.

Return type: Integer

**get_lsb()**
Returns the low part of the bytearray (component 1).

Returns: low part of the bytearray.

Return type: Integer

**address**
Bytearray. Bytearray representation of this XBee16BitAddress.

class digi.xbee.models.address.XBee64BitAddress(address)
Bases: object

This class represents a 64-bit address (also known as MAC address).

The 64-bit address is a unique device address assigned during manufacturing. This address is unique to each physical device.

Class constructor. Instantiates a new XBee64BitAddress object with the provided parameters.

Parameters: address (Bytearray) – the XBe 64-bit address as byte array.

Raise: ValueError: if address is None or its length less than 1 or greater than 8.

PATTERN = '\^0\[xX\]?\[0-9a-fA-F\]{1,16}$\'
64-bit address string pattern.

COORDINATOR_ADDRESS = <digi.xbee.models.address.XBee64BitAddress object>
0000000000000000.

Type: 64-bit address reserved for the coordinator (value

BROADCAST_ADDRESS = <digi.xbee.models.address.XBee64BitAddress object>
000000000000FFFF.

Type: 64-bit broadcast address (value

UNKNOWN_ADDRESS = <digi.xbee.models.address.XBee64BitAddress object>
FFFFFFFFFFFFFFFF.

Type: 64-bit unknown address (value

classmethod from_hex_string(address)
Class constructor. Instantiates a new XBee64BitAddress object from the provided hex string.

Parameters: address (String) – The XBe 64-bit address as a string.

Raises: ValueError – if the address’ length is less than 1 or does not match with the pattern: \(0\[xX\]?\[0-9a-fA-F\]{1,16}\).

classmethod from_bytes(*args)
Class constructor. Instantiates a new XBee64BitAddress object from the provided bytes.

Parameters: args (8 Integers) – 8 integers that represent the bytes 1 to 8 of this XBee64BitAddress.

Raises: ValueError – if the amount of arguments is not 8 or if any of the arguments is not between 0 and 255.
classmethod is_valid(address)
Checks if the provided hex string is a valid 64-bit address.

Parameters address (String or Bytearray) – The XBee 64-bit address as a string or bytestring.

Returns True for a valid 64-bit address, False otherwise.

Return type Boolean

address
Bytearray. Bytearray representation of this XBee64BitAddress.

class digi.xbee.models.address.XBeeIMEIAddress(address)
Bases: object

This class represents an IMEI address used by cellular devices.

This address is only applicable for Cellular protocol.

Class constructor. Instantiates a new :XBeeIMEIAddress object with the provided parameters.

Parameters address (Bytearray) – The XBee IMEI address as byte array.

Raises
• ValueError – if address is None.
• ValueError – if length of address greater than 8.

PATTERN = '^\d{0,15}$'
IMEI address string pattern.

classmethod from_string(address)
Class constructor. Instantiates a new :XBeeIMEIAddress object from the provided string.

Parameters address (String) – The XBee IMEI address as a string.

Raises
• ValueError – if address is None.
• ValueError – if address does not match the pattern: ^\d{0,15}$.

classmethod is_valid(address)
Checks if the provided hex string is a valid IMEI.

Parameters address (String or Bytearray) – The XBee IMEI address as a string or bytestring.

Returns True for a valid IMEI, False otherwise.

Return type Boolean

address
String. String representation of this XBeeIMEIAddress.

digi.xbee.models.message module

class digi.xbee.models.message.XBeeMessage(data, remote_device, timestamp, broadcast=False)
Bases: object

This class represents a XBee message, which is formed by a RemoteXBeeDevice (the sender) and some data (the data sent) as a bytestring.
Class constructor.

Parameters

- **data** (Bytearray) – the data sent.
- **remote_device** (RemoteXBeeDevice) – the sender.
- **broadcast** (Boolean, optional, default=``False``) – flag indicating whether the message is broadcast (True) or not (False). Optional.
- **timestamp** – instant of time when the message was received.

to_dict()

Returns the message information as a dictionary.

data

Bytearray. Bytearray containing the data of the message.

remote_device

RemoteXBeeDevice. The device that has sent the message.

is_broadcast

Boolean. True to indicate that the message is broadcast, False otherwise.

timestamp

Integer. Instant of time when the message was received.

class digi.xbee.models.message.ExplicitXBeeMessage(data, remote_device, timestamp, source_endpoint, dest_endpoint, cluster_id, profile_id, broadcast=False)

Bases: digi.xbee.models.message.XBeeMessage

This class represents an Explicit XBee message, which is formed by all parameters of a common XBee message and: Source endpoint, destination endpoint, cluster ID, profile ID.

Class constructor.

Parameters

- **data** (Bytearray) – the data sent.
- **remote_device** (RemoteXBeeDevice) – the sender device.
- **timestamp** – instant of time when the message was received.
- **source_endpoint** (Integer) – source endpoint of the message. 1 byte.
- **dest_endpoint** (Integer) – destination endpoint of the message. 1 byte.
- **cluster_id** (Integer) – cluster id of the message. 2 bytes.
- **profile_id** (Integer) – profile id of the message. 2 bytes.
- **broadcast** (Boolean, optional, default=``False``) – flag indicating whether the message is broadcast (True) or not (False). Optional.

to_dict()

Returns the message information as a dictionary.

source_endpoint

Integer. The source endpoint of the message

dest_endpoint

Integer. The destination endpoint of the message
cluster_id  
    Integer. The Cluster ID of the message.

profile_id  
    Integer. The profile ID of the message.

data  
    Returns the data of the message.

    Returns the data of the message.

    Return type Bytearray

is_broadcast  
    Returns whether the message is broadcast or not.

    Returns True if the message is broadcast, False otherwise.

    Return type Boolean

remote_device  
    Returns the device which has sent the message.

    Returns the device which has sent the message.

    Return type RemoteXBeeDevice

timestamp  
    Returns the moment when the message was received as a time.time() function returned value.

    Returns the returned value of using time.time() function when the message was received.

    Return type Float

class digi.xbee.models.message.IPMessage(ip_addr, source_port, dest_port, protocol, data)  
Bases: object

This class represents an IP message containing the IP address the message belongs to, the source and destination ports, the IP protocol, and the content (data) of the message.

Class constructor.

Parameters
    • ip_addr (ipaddress.IPv4Address) – The IP address the message comes from.
    • source_port (Integer) – TCP or UDP source port of the transmission.
    • dest_port (Integer) – TCP or UDP destination port of the transmission.
    • protocol (IPProtocol) – IP protocol used in the transmission.
    • data (Bytearray) – the data sent.

Raises
    • ValueError – if ip_addr is None.
    • ValueError – if protocol is None.
    • ValueError – if data is None.
    • ValueError – if source_port is less than 0 or greater than 65535.
    • ValueError – if dest_port is less than 0 or greater than 65535.

to_dict()  
    Returns the message information as a dictionary.
ip_addr
ipaddress.IPv4Address. The IPv4 address this message is associated to.

source_port
Integer. The source port of the transmission.

dest_port
Integer. The destination port of the transmission.

protocol
IPProtocol. The protocol used in the transmission.

data
Bytarray. Bytarray containing the data of the message.

class digi.xbee.models.message.SMSMessage(phone_number, data)
Bases: object
This class represents an SMS message containing the phone number that sent the message and the content (data) of the message.
This class is used within the library to read SMS sent to Cellular devices.
Class constructor. Instantiates a new SMSMessage object with the provided parameters.

Parameters

- phone_number (String) – The phone number that sent the message.
- data (String) – The message text.

Raises

- ValueError – if phone_number is None.
- ValueError – if data is None.
- ValueError – if phone_number is not a valid phone number.

to_dict()
Returns the message information as a dictionary.

phone_number
String. The phone number that sent the message.

data
String. The data of the message.

class digi.xbee.models.message.UserDataRelayMessage(local_interface, data)
Bases: object
This class represents a user data relay message containing the source interface and the content (data) of the message.

See also:

XBeeLocalInterface

Class constructor. Instantiates a new UserDataRelayMessage object with the provided parameters.

Parameters

- local_interface (XBeeLocalInterface) – The source XBee local interface.
• **data** (*Bytearray*) – Byte array containing the data of the message.

**Raises** **ValueError** – if `relay_interface` is `None`.

**See also:**

`XBeeLocalInterface`

to_dict()

Returns the message information as a dictionary.

**local_interface**

*XBeeLocalInterface*. Source interface that sent the message.

**data**

*Bytearray*. The data of the message.

**digi.xbee.models.options module**

class **digi.xbee.models.options.ReceiveOptions**

**Bases:** `enum.Enum`

This class lists all the possible options that have been set while receiving an XBee packet.

The receive options are usually set as a bitfield meaning that the options can be combined using the ‘|’ operand.

**Values:**

- **ReceiveOptions.NONE** = 0
- **ReceiveOptions.PACKET_ACKNOWLEDGED** = 1
- **ReceiveOptions.BROADCAST_PACKET** = 2
- **ReceiveOptions.APS_ENCRYPTED** = 32
- **ReceiveOptions.SENT_FROM_END_DEVICE** = 64

**NONE** = 0

No special receive options.

**PACKET_ACKNOWLEDGED** = 1

Packet was acknowledged.

Not valid for WiFi protocol.

**BROADCAST_PACKET** = 2

Packet was a broadcast packet.

Not valid for WiFi protocol.

**APS_ENCRYPTED** = 32

Packet encrypted with APS encryption.

Only valid for ZigBee XBee protocol.
SENT_FROM_END_DEVICE = 64
Packet was sent from an end device (if known).
Only valid for ZigBee XBee protocol.

```python
class digi.xbee.models.options.TransmitOptions
Bases: enum.Enum

This class lists all the possible options that can be set while transmitting an XBee packet.

The transmit options are usually set as a bitfield meaning that the options can be combined using the ‘|’ operand.

Not all options are available for all cases, that’s why there are different names with same values. In each moment, you must be sure that the option your are going to use, is a valid option in your context.

Values:
TransmitOptions.NONE = 0
TransmitOptions.DISABLE_ACK = 1
TransmitOptions.DONT_ATTEMPT_RD = 2
TransmitOptions.USE_BROADCAST_PAN_ID = 4
TransmitOptions.ENABLE_MULTICAST = 8
TransmitOptions.ENABLE_APS_ENCRYPTION = 32
TransmitOptions.USE_EXTENDED_TIMEOUT = 64
TransmitOptions.REPEATER_MODE = 128
TransmitOptions.DIGIMESH_MODE = 192
```

**NONE** = 0
No special transmit options.

**DISABLE_ACK** = 1
Disables acknowledgments on all unicasts.
Only valid for DigiMesh, 802.15.4 and Point-to-multipoint protocols.

**DISABLE_RETRIES_AND_REPAIR** = 1
Disables the retries and router repair in the frame.
Only valid for ZigBee protocol.

**DONT_ATTEMPT_RD** = 2
Doesn’t attempt Route Discovery.
Disables Route Discovery on all DigiMesh unicasts.
Only valid for DigiMesh protocol.

**USE_BROADCAST_PAN_ID** = 4
Sends packet with broadcast {code PAN ID}. Packet will be sent to all devices in the same channel ignoring the {code PAN ID}.
It cannot be combined with other options.
Only valid for 802.15.4 XBee protocol.

**ENABLE_UNICAST_NACK** = 4
Enables unicast NACK messages.
NACK message is enabled on the packet.
Only valid for DigiMesh 868/900 protocol.

**ENABLE_UNICAST_TRACE_ROUTE = 4**
Enables unicast trace route messages.
Trace route is enabled on the packets.
Only valid for DigiMesh 868/900 protocol.

**ENABLE_MULTICAST = 8**
Enables multicast transmission request.
Only valid for ZigBee XBee protocol.

**ENABLE_APS_ENCRYPTION = 32**
Enables APS encryption, only if `EE=1`.
Enabling APS encryption decreases the maximum number of RF payload bytes by 4 (below the value reported by `NP`).
Only valid for ZigBee XBee protocol.

**USE_EXTENDED_TIMEOUT = 64**
Uses the extended transmission timeout.
Setting the extended timeout bit causes the stack to set the extended transmission timeout for the destination address.
Only valid for ZigBee XBee protocol.

**POINT_MULTIPOINT_MODE = 64**
Transmission is performed using point-to-Multipoint mode.
Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

**REPEATER_MODE = 128**
Transmission is performed using repeater mode.
Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

**DIGIMESH_MODE = 192**
Transmission is performed using DigiMesh mode.
Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

class digi.xbee.models.options.RemoteATCmdOptions
Bases: enum.Enum

This class lists all the possible options that can be set while transmitting a remote AT Command.
These options are usually set as a bitfield meaning that the options can be combined using the ‘|’ operand.

Values:
RemoteATCmdOptions.NONE = 0
RemoteATCmdOptions.DISABLE_ACK = 1
RemoteATCmdOptions.APPLY_CHANGES = 2
RemoteATCmdOptions.EXTENDED_TIMEOUT = 64
NONE = 0
No special transmit options

DISABLE_ACK = 1
Disables ACK

APPLY_CHANGES = 2
Applies changes in the remote device.
If this option is not set, AC command must be sent before changes will take effect.

EXTENDED_TIMEOUT = 64
Uses the extended transmission timeout
Setting the extended timeout bit causes the stack to set the extended transmission timeout for the destination address.
Only valid for ZigBee XBee protocol.

class digi.xbee.models.options.SendDataRequestOptions(code, description)
Bases: enum.Enum

Enumerates the different options for the SendDataRequestPacket.

Values:
SendDataRequestOptions.OVERWRITE = (0, 'Overwrite')
SendDataRequestOptions.ARCHIVE = (1, 'Archive')
SendDataRequestOptions.APPEND = (2, 'Append')
SendDataRequestOptions.TRANSIENT = (3, 'Transient data (do not store)')

code
Integer. The send data request option code.

description
String. The send data request option description.

class digi.xbee.models.options.DiscoveryOptions(code, description)
Bases: enum.Enum

Enumerates the different options used in the discovery process.

Values:
DiscoveryOptions.APPEND_DD = (1, 'Append device type identifier (DD)')
DiscoveryOptions.DISCOVER_MYSELF = (2, 'Local device sends response frame')
DiscoveryOptions.APPEND_RSSI = (4, 'Append RSSI (of the last hop)')

APPEND_DD = (1, 'Append device type identifier (DD)')
Append device type identifier (DD) to the discovery response.

Valid for the following protocols:
• DigiMesh
• Point-to-multipoint (Digi Point)
• ZigBee

**DISCOVER_MYSELF = (2, 'Local device sends response frame')**

Local device sends response frame when discovery is issued.

**Valid for the following protocols:**

• DigiMesh
• Point-to-multipoint (Digi Point)
• ZigBee
• 802.15.4

**APPEND_RSSI = (4, 'Append RSSI (of the last hop)')**

Append RSSI of the last hop to the discovery response.

**Valid for the following protocols:**

• DigiMesh
• Point-to-multipoint (Digi Point)

code

Integer. The discovery option code.

description

String. The discovery option description.

class digi.xbee.models.options.XBeeLocalInterface(code, description)

**Bases:** enum.Enum

Enumerates the different interfaces for theUserDataRelayPacket and UserDataRelayOutputPacket.

Inherited properties:

**name** (String): the name (id) of the XBee local interface.

**value** (String): the value of the XBee local interface.

Values:

**XBeeLocalInterface.SERIAL** = (0, ‘Serial port (UART when in API mode, or SPI interface)’)

**XBeeLocalInterface.BLUETOOTH** = (1, ‘BLE API interface (on XBee devices which support BLE)’)

**XBeeLocalInterface.MICROPYTHON** = (2, ‘MicroPython’)

**XBeeLocalInterface.UNKNOWN** = (255, ‘Unknown interface’)

code

Integer. The XBee local interface code.

description

String. The XBee local interface description.

class digi.xbee.models.options.RegisterKeyOptions(code, description)

**Bases:** enum.Enum

This class lists all the possible options that have been set while receiving an XBee packet.
The receive options are usually set as a bitfield meaning that the options can be combined using the ‘|’ operand.

Values:
- `RegisterKeyOptions.LINK_KEY` = (0, ‘Key is a Link Key (KY on joining node)’)
- `RegisterKeyOptions.INSTALL_CODE` = (1, ‘Key is an Install Code (I? on joining node, DC must be set to 1 on joiner)’)
- `RegisterKeyOptions.UNKNOWN` = (255, ‘Unknown key option’)

```python
code
Integer. The register key option code.
description
String. The register key option description.
class digi.xbee.models.options.SocketOption(code, description)
    Bases: enum.Enum

Enumerates the different Socket Options.
```

Values:
- `SocketOption.TLS_PROFILE` = (0, ‘TLS Profile’)
- `SocketOption.UNKNOWN` = (255, ‘Unknown’)

```python
code
Integer. The Socket Option code.
description
String. The Socket Option description.
class digi.xbee.models.protocol.XBeeProtocol(code, description)
    Bases: enum.Enum

Enumerates the available XBee protocols. The XBee protocol is determined by the combination of hardware and firmware of an XBee device.

Inherited properties:
- `name` (String): the name (id) of this XBeeProtocol.
- `value` (String): the value of this XBeeProtocol.

Values:
- `XBeeProtocol.ZIGBEE` = (0, ‘ZigBee’)
- `XBeeProtocol.RAW_802_15_4` = (1, ‘802.15.4’)
- `XBeeProtocol.XBEE_WIFI` = (2, ‘Wi-Fi’)
```
XBeeProtocol.DIGI_MESH = (3, ‘DigiMesh’)
XBeeProtocol.XCITE = (4, ‘XCite’)
XBeeProtocol.XTEND = (5, ‘XTend (Legacy)’)
XBeeProtocol.XTEND_DM = (6, ‘XTend (DigiMesh)’)
XBeeProtocol.SMART_ENERGY = (7, ‘Smart Energy’)
XBeeProtocol.DIGI_POINT = (8, ‘Point-to-multipoint’)
XBeeProtocol.ZNET = (9, ‘ZNet 2.5’)
XBeeProtocol.XC = (10, ‘XSC’)
XBeeProtocol.XLR = (11, ‘XLR’)
XBeeProtocol.XLR_DM = (12, ‘XLR’)
XBeeProtocol.SX = (13, ‘XBee SX’)
XBeeProtocol.XLR_MODULE = (14, ‘XLR Module’)
XBeeProtocol.CELLULAR = (15, ‘Cellular’)
XBeeProtocol.CELLULAR_NBIOT = (16, ‘Cellular NB-IoT’)
XBeeProtocol.UNKNOWN = (99, ‘Unknown’)

code
Integer. XBee protocol code.
description
String. XBee protocol description.

class digi.xbee.models.protocol.IPProtocol (code, description)
Bases: enum.Enum

Enumerates the available network protocols.

Inherited properties:

name (String): the name (id) of this IPProtocol.
value (String): the value of this IPProtocol.

Values:
IPProtocol.UDP = (0, ‘UDP’)
IPProtocol.TCP = (1, ‘TCP’)
IPProtocol.TCP_SSL = (4, ‘TLS’)

code
IP protocol code.
Type Integer
description
IP protocol description.
Type String

class digi.xbee.models.protocol.Role (identifier, description)
Bases: enum.Enum
Enumerates the available roles for an XBee.

Inherited properties:

- **name** (String): the name (id) of this Role.
- **value** (String): the value of this Role.

Values:

- `Role.COORDINATOR = (0, ‘Coordinator’)
- `Role.ROUTER = (1, ‘Router’)
- `Role.END_DEVICE = (2, ‘End device’)
- `Role.UNKNOWN = (3, ‘Unknown’)

**id**

Gets the identifier of the role.

- **Returns** the role identifier.
- **Return type** Integer

**description**

Gets the description of the role.

- **Returns** the role description.
- **Return type** String

---

digi.xbee.models.status module

class digi.xbee.models.status.ATCommandStatus (code, description)

Bases: enum.Enum

This class lists all the possible states of an AT command after executing it.

Inherited properties:

- **name** (String): the name (id) of the ATCommandStatus.
- **value** (String): the value of the ATCommandStatus.

Values:

- `ATCommandStatus.OK = (0, ‘Status OK’)
- `ATCommandStatus.ERROR = (1, ‘Status Error’)
- `ATCommandStatus.INVALID_COMMAND = (2, ‘Invalid command’)
- `ATCommandStatus.INVALID_PARAMETER = (3, ‘Invalid parameter’)
- `ATCommandStatus.TX_FAILURE = (4, ‘TX failure’)
- `ATCommandStatus.UNKNOWN = (255, ‘Unknown status’)

---

2.5. API reference 137
code
Integer. The AT command status code.

description
String. The AT command status description.

class digi.xbee.models.status.DiscoveryStatus (code, description)
Bases: enum.Enum

This class lists all the possible states of the discovery process.

Inherited properties:

name (String): The name of the DiscoveryStatus.
value (Integer): The ID of the DiscoveryStatus.

Values:
DiscoveryStatus.NO_DISCOVERY_OVERHEAD = (0, ‘No discovery overhead’)
DiscoveryStatus.ADDRESS_DISCOVERY = (1, ‘Address discovery’)
DiscoveryStatus.ROUTE_DISCOVERY = (2, ‘Route discovery’)
DiscoveryStatus.ADDRESS_AND_ROUTE = (3, ‘Address and route’)
DiscoveryStatus.EXTENDED_TIMEOUT_DISCOVERY = (64, ‘Extended timeout discovery’)
DiscoveryStatus.UNKNOWN = (255, ‘Unknown’)

code
Integer. The discovery status code.

description
String. The discovery status description.

class digi.xbee.models.status.TransmitStatus (code, description)
Bases: enum.Enum

This class represents all available transmit status.

Inherited properties:

name (String): the name (id) of the TransmitStatus.
value (String): the value of the TransmitStatus.

Values:
TransmitStatus.SUCCESS = (0, ‘Success.’)
TransmitStatus.NO_ACK = (1, ‘No acknowledgement received.’)
TransmitStatus.CCA_FAILURE = (2, ‘CCA failure.’)
TransmitStatus.PURGED = (3, ‘Transmission purged, it was attempted before stack was up.’)
TransmitStatus.WIFI_PHYSICAL_ERROR = (4, ‘Physical error occurred on the interface with the WiFi transceiver.’)
TransmitStatus.INVALID_DESTINATION = (21, ‘Invalid destination endpoint.’)
TransmitStatus.NO_BUFFERS = (24, ‘No buffers.’)
TransmitStatus.NETWORK_ACK_FAILURE = (33, ‘Network ACK Failure.’)
TransmitStatus.NOT_JOINED_NETWORK = (34, ‘Not joined to network.’)
TransmitStatus.SELF_ADDRESSED = (35, ‘Self-addressed.’)
TransmitStatus.ADDRESS_NOT_FOUND = (36, ‘Address not found.’)
TransmitStatus.ROUTE_NOT_FOUND = (37, ‘Route not found.’)
TransmitStatus.BROADCAST_FAILED = (38, ‘Broadcast source failed to hear a neighbor relay the message.’)
TransmitStatus.INVALID_BINDING_TABLE_INDEX = (43, ‘Invalid binding table index.’)
TransmitStatus.INVALID_ENDPOINT = (44, ‘Invalid endpoint’)
TransmitStatus.BROADCAST_ERROR_APS = (45, ‘Attempted broadcast with APS transmission.’)
TransmitStatus.BROADCAST_ERROR_APS_EE0 = (46, ‘Attempted broadcast with APS transmission, but EE=0.’)
TransmitStatus.SOFTWARE_ERROR = (49, ‘A software error occurred.’)
TransmitStatus.RESOURCE_ERROR = (50, ‘Resource error lack of free buffers, timers, etc.’)
TransmitStatus.PAYLOAD_TOO_LARGE = (116, ‘Data payload too large.’)
TransmitStatus.INDIRECT_MESSAGE_UNREQUESTED = (117, ‘Indirect message unrequested’)
TransmitStatus.SOCKET_CREATION_FAILED = (118, ‘Attempt to create a client socket failed.’)
TransmitStatus.IP_PORT_NOT_EXIST = (119, “TCP connection to given IP address and port doesn’t exist. Source port is non-zero so that a new connection is not attempted.”)
TransmitStatus.UDP_SRC_PORT_NOT_MATCH_LISTENING_PORT = (120, “Source port on a UDP transmission doesn’t match a listening port on the transmitting module.”)
TransmitStatus.TCP_SRC_PORT_NOT_MATCH_LISTENING_PORT = (121, “Source port on a TCP transmission doesn’t match a listening port on the transmitting module.”)
TransmitStatus.INVALID_IP_ADDRESS = (122, ‘Destination IPv4 address is not valid.’)
TransmitStatus.INVALID_IP_PROTOCOL = (123, ‘Protocol on an IPv4 transmission is not valid.’)
TransmitStatus.RELAY_INTERFACE_INVALID = (124, “Destination interface on a User Data Relay Frame doesn’t exist.”)
TransmitStatus.RELAY_INTERFACE_REJECTED = (125, ‘Destination interface on a User Data Relay Frame exists, but the interface is not accepting data.’)
TransmitStatus.SOCKET_CONNECTION_REFUSED = (128, ‘Destination server refused the connection.’)
TransmitStatus.SOCKET_CONNECTION_LOST = (129, ‘The existing connection was lost before the data was sent.’)
TransmitStatus.SOCKET_ERROR_NO_SERVER = (130, ‘The attempted connection timed out.’)
TransmitStatus.SOCKET_ERROR_CLOSED = (131, ‘The existing connection was closed.’)
TransmitStatus.SOCKET_ERROR_UNKNOWN_SERVER = (132, ‘The server could not be found.’)
TransmitStatus.SOCKET_ERROR_UNKNOWN_ERROR = (133, ‘An unknown error occurred.’)
TransmitStatus.INVALID_TLS_CONFIGURATION = (134, “TLS Profile on a 0x23 API request doesn’t exist, or one or more certificates is not valid.”)
TransmitStatus.KEY_NOTAUTHORIZED = (187, ‘Key not authorized.’)
TransmitStatus.UNKNOWN = (255, ‘Unknown.’)

code
Integer. The transmit status code.

description
String. The transmit status description.
class digi.xbee.models.status.ModemStatus(code, description)
Bases: `enum.Enum`

Enumerates the different modem status events. This enumeration list is intended to be used within the `ModemStatusPacket` packet.

Values:

- `ModemStatus.HARDWARE_RESET` = (0, ‘Device was reset’)
- `ModemStatus.WATCHDOG_TIMER_RESET` = (1, ‘Watchdog timer was reset’)
- `ModemStatus.JOINED_NETWORK` = (2, ‘Device joined to network’)
- `ModemStatus.DISASSOCIATED` = (3, ‘Device disassociated’)
- `ModemStatus.ERROR_SYNCHRONIZATION_LOST` = (4, ‘Configuration error/synchronization lost’)
- `ModemStatus.COORDINATOR_REALIGNMENT` = (5, ‘Coordinator realignment’)
- `ModemStatus.COORDINATOR_STARTED` = (6, ‘The coordinator started’)
- `ModemStatus.NETWORK_SECURITY_KEY_UPDATED` = (7, ‘Network security key was updated’)
- `ModemStatus.NETWORK_WOKE_UP` = (11, ‘Network Woke Up’)
- `ModemStatus.NETWORK_WENT_TO_SLEEP` = (12, ‘Network Went To Sleep’)
- `ModemStatus.VOLTAGE_SUPPLY_LIMIT_EXCEEDED` = (13, ‘Voltage supply limit exceeded’)
- `ModemStatus.REMOTE_MANAGER_CONNECTED` = (14, ‘Remote Manager connected’)
- `ModemStatus.REMOTE_MANAGER_DISCONNECTED` = (15, ‘Remote Manager disconnected’)
- `ModemStatus.MODEM_CONFIG_CHANGED_WHILE.Joining` = (17, ‘Modem configuration changed while joining’)
- `ModemStatus.BLUETOOTH_CONNECTED` = (50, ‘A Bluetooth connection has been made and API mode has been unlocked.’)
- `ModemStatus.BLUETOOTH_DISCONNECTED` = (51, ‘An unlocked Bluetooth connection has been disconnected.’)
- `ModemStatus.BANDMASK_CONFIGURATION_ERROR` = (52, ‘LTE-M/NB-IoT bandmask configuration has failed.’)
- `ModemStatus.ERROR_STACK` = (128, ‘Stack error’)
- `ModemStatus.ERROR_AP_NOT_CONNECTED` = (130, ‘Send/join command issued without connecting from AP’)
- `ModemStatus.ERROR_AP_NOT_FOUND` = (131, ‘Access point not found’)
- `ModemStatus.ERROR_PSK_NOT_CONFIGURED` = (132, ‘PSK not configured’)
- `ModemStatus.ERROR_SSID_NOT_FOUND` = (135, ‘SSID not found’)
- `ModemStatus.ERROR_FAILED_JOIN_SECURITY` = (136, ‘Failed to join with security enabled’)
- `ModemStatus.ERROR_INVALID_CHANNEL` = (138, ‘Invalid channel’)
- `ModemStatus.ERROR_FAILED_JOIN_AP` = (142, ‘Failed to join access point’)
- `ModemStatus.UNKNOWN` = (255, ‘UNKNOWN’)

**code**

Integer. The modem status code.

**description**

String. The modem status description.

class `digi.xbee.models.status.PowerLevel` *(code, description)*

Bases: `enum.Enum`
Enumerates the different power levels. The power level indicates the output power value of a radio when transmitting data.

Values:
- `PowerLevel.LEVEL_LOWEST` = (0, ‘Lowest’)
- `PowerLevel.LEVEL_LOW` = (1, ‘Low’)
- `PowerLevel.LEVEL_MEDIUM` = (2, ‘Medium’)
- `PowerLevel.LEVEL_HIGH` = (3, ‘High’)
- `PowerLevel.LEVEL_HIGHEST` = (4, ‘Highest’)
- `PowerLevel.LEVEL_UNKNOWN` = (255, ‘Unknown’)

code
- Integer. The power level code.

description
- String. The power level description.

class digi.xbee.models.status.AssociationIndicationStatus (code, description)

Bases: enum.Enum

Enumerates the different association indication statuses.

Values:
- `AssociationIndicationStatus.SUCCESSFULLY_JOINED` = (0, ‘Successfully formed or joined a network.’)
- `AssociationIndicationStatus.AS_TIMEOUT` = (1, ‘Active Scan Timeout.’)
- `AssociationIndicationStatus.AS_NO_PANS_FOUND` = (2, ‘Active Scan found no PANS.’)
- `AssociationIndicationStatus.AS_ASSOCIATION_NOT_ALLOWED` = (3, ‘Active Scan found PAN, but the CoordinatorAllowAssociation bit is not set.’)
- `AssociationIndicationStatus.AS_BEACONS_NOT_SUPPORTED` = (4, ‘Active Scan found PAN, but Coordinator and End Device are not configured to support beacons.’)
- `AssociationIndicationStatus.AS_ID_DOESNT_MATCH` = (5, ‘Active Scan found PAN, but the Coordinator ID parameter does not match the ID parameter of the End Device.’)
- `AssociationIndicationStatus.AS_CHANNEL_DOESNT_MATCH` = (6, ‘Active Scan found PAN, but the Coordinator CH parameter does not match the CH parameter of the End Device.’)
- `AssociationIndicationStatus.ENERGY_SCAN_TIMEOUT` = (7, ‘Energy Scan Timeout.’)
- `AssociationIndicationStatus.COORDINATOR_START_REQUEST_FAILED` = (8, ‘Coordinator start request failed.’)
- `AssociationIndicationStatus.COORDINATOR_INVALID_PARAMETER` = (9, ‘Coordinator could not start due to invalid parameter.’)
- `AssociationIndicationStatus.COORDINATOR_REALIGNMENT` = (10, ‘Coordinator Realignment is in progress.’)
- `AssociationIndicationStatus.AR_NOT_SENT` = (11, ‘Association Request not sent.’)
- `AssociationIndicationStatus.AR_TIMED_OUT` = (12, ‘Association Request timed out - no reply was received.’)
- `AssociationIndicationStatus.AR_INVALID_PARAMETER` = (13, ‘Association Request had an Invalid Parameter.’)
AssociationIndicationStatus.AR_CHANNEL_ACCESS_FAILURE = (14, ‘Association Request Channel Access Failure. Request was not transmitted - CCA failure.’)
AssociationIndicationStatus.AR_COORDINATOR_ACK_WASNT_RECEIVED = (15, ‘Remote Coordinator did not send an ACK after Association Request was sent.’)
AssociationIndicationStatus.AR_COORDINATOR_DIDNT_REPLY = (16, ‘Remote Coordinator did not reply to the Association Request, but an ACK was received after sending the request.’)
AssociationIndicationStatus.SYNCHRONIZATION_LOST = (18, ‘Sync-Loss - Lost synchronization with a Beaconing Coordinator.’)
AssociationIndicationStatus.DISASSOCIATED = (19, ‘Disassociated - No longer associated to Coordinator.’)
AssociationIndicationStatus.NO_PANS_FOUND = (33, ‘Scan found no PANs.’)
AssociationIndicationStatus.NO_PANS_WITH_ID_FOUND = (34, ‘Scan found no valid PANs based on current SC and ID settings.’)
AssociationIndicationStatus.NJ_EXPIRED = (35, ‘Valid Coordinator or Routers found, but they are not allowing joining (NJ expired).’)
AssociationIndicationStatus.NO_JOINABLE_BEACONS_FOUND = (36, ‘No joinable beacons were found.’)
AssociationIndicationStatus.UNEXPECTED_STATE = (37, ‘Unexpected state, node should not be attempting to join at this time.’)
AssociationIndicationStatus.JOIN_FAILED = (39, ‘Node Joining attempt failed (typically due to incompatible security settings).’)
AssociationIndicationStatus.COORDINATOR_START_FAILED = (42, ‘Coordinator Start attempt failed.’)
AssociationIndicationStatus.CHECKING_FOR_COORDINATOR = (43, ‘Checking for an existing coordinator.’)
AssociationIndicationStatus.NETWORK_LEAVE_FAILED = (44, ‘Attempt to leave the network failed.’)
AssociationIndicationStatus.DEVICE_DIDNT_RESPOND = (171, ‘Attempted to join a device that did not respond.’)
AssociationIndicationStatus.UNSECURED_KEY_RECEIVED = (172, ‘Secure join error - network security key received unsecured.’)
AssociationIndicationStatus.KEY_NOT_RECEIVED = (173, ‘Secure join error - network security key not received.’)
AssociationIndicationStatus.INVALID_SECURITY_KEY = (175, ‘Secure join error - joining device does not have the right preconfigured link key.’)
AssociationIndicationStatus.SCANNING_NETWORK = (255, ‘Scanning for a network/Attempting to associate.’)

**code**

Integer. The association indication status code.

description

String. The association indication status description.

class digi.xbee.models.status.CellularAssociationIndicationStatus (code, description)

Bases: enum.Enum

Enumerates the different association indication statuses for the Cellular protocol.
Values:

- **CellularAssociationIndicationStatus.SUCCESSFULLY_CONNECTED** = (0, 'Connected to the Internet.')
- **CellularAssociationIndicationStatus.REGISTERING_CELLULAR_NETWORK** = (34, 'Registering to cellular network')
- **CellularAssociationIndicationStatus.CONNECTING_INTERNET** = (35, 'Connecting to the Internet')
- **CellularAssociationIndicationStatus.MODEM_FIRMWARE_CORRUPT** = (36, 'The cellular component requires a new firmware image.')
- **CellularAssociationIndicationStatus.REGISTRATION_DENIED** = (37, 'Cellular network registration was denied.')
- **CellularAssociationIndicationStatus.AIRPLANE_MODE** = (42, 'Airplane mode is active.')
- **CellularAssociationIndicationStatus.USB_DIRECT** = (43, 'USB Direct mode is active.')
- **CellularAssociationIndicationStatus.PSM_LOWPOWER** = (44, 'The cellular component is in the PSM low-power state.')
- **CellularAssociationIndicationStatus.BYPASS_MODE** = (47, 'Bypass mode active')
- **CellularAssociationIndicationStatus.INITIALIZING** = (255, 'Initializing')

**code**

Integer. The cellular association indication status code.

**description**

String. The cellular association indication status description.

**class** digi.xbee.models.status.DeviceCloudStatus(*code*, *description*)

Bases: enum.Enum

Enumerates the different Device Cloud statuses.

Values:

- **DeviceCloudStatus.SUCCESS** = (0, 'Success')
- **DeviceCloudStatus.BAD_REQUEST** = (1, 'Bad request')
- **DeviceCloudStatus.RESPONSE_UNAVAILABLE** = (2, 'Response unavailable')
- **DeviceCloudStatus.DEVICE_CLOUD_ERROR** = (3, 'Device Cloud error')
- **DeviceCloudStatus.CANCELED** = (32, 'Device Request canceled by user')
- **DeviceCloudStatus.TIME_OUT** = (33, 'Session timed out')
- **DeviceCloudStatus.UNKNOWN_ERROR** = (64, 'Unknown error')

**code**

Integer. The Device Cloud status code.

**description**

String. The Device Cloud status description.

**class** digi.xbee.models.status.FrameError(*code*, *description*)

Bases: enum.Enum

Enumerates the different frame errors.
Values:

FrameError.INVALID_TYPE = (2, 'Invalid frame type’)
FrameError.INVALID_LENGTH = (3, 'Invalid frame length’)
FrameError.INVALID_CHECKSUM = (4, 'Erroneous checksum on last frame’)
FrameError.PAYLOAD_TOO_BIG = (5, 'Payload of last API frame was too big to fit into a buffer’)
FrameError.STRING_ENTRY_TOO_BIG = (6, 'String entry was too big on last API frame sent’)
FrameError.WRONG_STATE = (7, ‘Wrong state to receive frame’)
FrameError.WRONG_REQUEST_ID = (8, “Device request ID of device response didn’t match the number in the request”)

code
Integer. The frame error code.
description
String. The frame error description.

class digi.xbee.models.status.WiFiAssociationIndicationStatus (code, description)
Bases: enum.Enum

Enumerates the different Wi-Fi association indication statuses.

Values:

WiFiAssociationIndicationStatus.SUCCESSFULLY_JOINED = (0, ‘Successfully joined to access point.’)
WiFiAssociationIndicationStatus.INITIALIZING = (1, ‘Initialization in progress.’)
WiFiAssociationIndicationStatus.INITIALIZED = (2, ‘Initialized, but not yet scanning.’)
WiFiAssociationIndicationStatus.DISCONNECTING = (19, ‘Disconnecting from access point.’)
WiFiAssociationIndicationStatus.SSID_NOT_CONFIGURED = (35, ‘SSID not configured’)
WiFiAssociationIndicationStatus.INVALID_KEY = (36, ‘Encryption key invalid (NULL or invalid length).’)
WiFiAssociationIndicationStatus.JOIN_FAILED = (39, ‘SSID found, but join failed.’)
WiFiAssociationIndicationStatus.WAITING_FOR_AUTH = (64, ‘Waiting for WPA or WPA2 authentication.’)
WiFiAssociationIndicationStatus.WAITING_FOR_IP = (65, ‘Joined to a network and waiting for IP address.’)
WiFiAssociationIndicationStatus.SETTING_UP_SOCKETS = (66, ‘Joined to a network and IP configured. Setting up listening sockets.’)
WiFiAssociationIndicationStatus.SCANNING_FOR_SSID = (255, ‘Scanning for the configured SSID.’)

code
Integer. The Wi-Fi association indication status code.
description
String. The Wi-Fi association indication status description.

class digi.xbee.models.status.NetworkDiscoveryStatus (code, description)
Bases: enum.Enum
Enumerates the different statuses of the network discovery process.

Values:

- `NetworkDiscoveryStatus.SUCCESS = (0, ‘Success’)`
- `NetworkDiscoveryStatus.ERROR_READ_TIMEOUT = (1, ‘Read timeout error’)`
- `NetworkDiscoveryStatus.ERROR_NET_DISCOVER = (2, ‘Error executing node discovery’)`
- `NetworkDiscoveryStatus.ERROR_GENERAL = (3, ‘Error while discovering network’)`
- `NetworkDiscoveryStatus.CANCEL = (4, ‘Discovery process cancelled’)`

code
- Integer. The network discovery status code.

description
- String. The network discovery status description.

class `digi.xbee.models.status.ZigbeeRegisterStatus`(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different statuses of the Zigbee Device Register process.

Values:

- `ZigbeeRegisterStatus.SUCCESS = (0, ‘Success’)`
- `ZigbeeRegisterStatus.KEY_TOO_LONG = (1, ‘Key too long’)`
- `ZigbeeRegisterStatus.ADDRESS_NOT_FOUND = (177, ‘Address not found in the key table’)`
- `ZigbeeRegisterStatus.INVALID_KEY = (178, ‘Key is invalid (00 and FF are reserved)’)`
- `ZigbeeRegisterStatus.INVALID_ADDRESS = (179, ‘Invalid address’)`
- `ZigbeeRegisterStatus.KEY_TABLE_FULL = (180, ‘Key table is full’)`
- `ZigbeeRegisterStatus.KEY_NOT_FOUND = (255, ‘Key not found’)`
- `ZigbeeRegisterStatus.UNKNOWN = (238, ‘Unknown’)`

code
- Integer. The Zigbee Device Register status code.

description
- String. The Zigbee Device Register status description.

class `digi.xbee.models.status.SocketStatus`(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different Socket statuses.

Values:

- `SocketStatus.SUCCESS = (0, ‘Operation successful’)`
- `SocketStatus.INVALID_PARAM = (1, ‘Invalid parameters’)`
- `SocketStatus.FAILED_TO_READ = (2, ‘Failed to retrieve option value’)`
- `SocketStatus.CONNECTION_IN_PROGRESS = (3, ‘Connection already in progress’)`
- `SocketStatus.ALREADY_CONNECTED = (4, ‘Already connected/bound/listening’)`

2.5. API reference
SocketStatus.UNKNOWN_ERROR = (5, ‘Unknown error’)
SocketStatus.BAD_SOCKET = (32, ‘Bad socket ID’)
SocketStatus.NOT_REGISTERED = (34, ‘Not registered to cell network’)
SocketStatus.INTERNAL_ERROR = (49, ‘Internal error’)
SocketStatus.RESOURCE_ERROR = (50, ‘Resource error: retry the operation later’)
SocketStatus.INVALID_PROTOCOL = (123, ‘Invalid protocol’)
SocketStatus.UNKNOWN = (255, ‘Unknown’)

code
Integer. The Socket status code.

description
String. The Socket status description.

class digi.xbee.models.status.SocketState (code, description)
    Bases: enum.Enum

    Enumerates the different Socket states.

    Values:
    SocketState.CONNECTED = (0, ‘Connected’)
    SocketState.FAILED_DNS = (1, ‘Failed DNS lookup’)
    SocketState.CONNECTION_REFUSED = (2, ‘Connection refused’)
    SocketState.TRANSPORT_CLOSED = (3, ‘Transport closed’)
    SocketState.TIMED_OUT = (4, ‘Timed out’)
    SocketState.INTERNAL_ERROR = (5, ‘Internal error’)
    SocketState.HOST_UNREACHABLE = (6, ‘Host unreachable’)
    SocketState.CONNECTION_LOST = (7, ‘Connection lost’)
    SocketState.UNKNOWN_ERROR = (8, ‘Unknown error’)
    SocketState.UNKNOWN_SERVER = (9, ‘Unknown server’)
    SocketState.RESOURCE_ERROR = (10, ‘Resource error’)
    SocketState.LISTENER_CLOSED = (11, ‘Listener closed’)
    SocketState.UNKNOWN = (255, ‘Unknown’)

code
Integer. The Socket state code.

description
String. The Socket state description.

class digi.xbee.models.status.SocketInfoState (code, description)
    Bases: enum.Enum

    Enumerates the different Socket info states.

    Values:
    SocketInfoState.ALLOCATED = (0, ‘Allocated’)

SocketInfoState.CONNECTING = (1, ‘Connecting’)  
SocketInfoState.CONNECTED = (2, ‘Connected’)  
SocketInfoState.LISTENING = (3, ‘Listening’)  
SocketInfoState.BOUND = (4, ‘Bound’)  
SocketInfoState.CLOSING = (5, ‘Closing’)  
SocketInfoState.UNKNOWN = (255, ‘Unknown’)  

**code**  
Integer. The Socket info state code.  

**description**  
String. The Socket info state description.  

---  

**digi.xbee.packets package**  

**Submodules**  

**digi.xbee.packets.aft module**  

**class** digi.xbee.packets.aft.ApiFrameType(code, description)  
**Bases:** enum.Enum  

This enumeration lists all the available frame types used in any XBee protocol.  

Inherited properties:  

- **name** (String): the name (id) of this ApiFrameType.  
- **value** (String): the value of this ApiFrameType.  

**Values:**  

- ApiFrameType.TX_64 = (0, ‘TX (Transmit) Request 64-bit address’)  
- ApiFrameType.TX_16 = (1, ‘TX (Transmit) Request 16-bit address’)  
- ApiFrameType.REMOTE_AT_COMMAND_REQUEST_WIFI = (7, ‘Remote AT Command Request (Wi-Fi)’)  
- ApiFrameType.AT_COMMAND = (8, ‘AT Command’)  
- ApiFrameType.AT_COMMAND_QUEUE = (9, ‘AT Command Queue’)  
- ApiFrameType.TRANSMIT_REQUEST = (16, ‘Transmit Request’)  
- ApiFrameType.EXPLICIT_ADDRESSING = (17, ‘Explicit Addressing Command Frame’)  
- ApiFrameType.REMOTE_AT_COMMAND_REQUEST = (23, ‘Remote AT Command Request’)  
- ApiFrameType.TX_SMS = (31, ‘TX SMS’)  
- ApiFrameType.TX_IPV4 = (32, ‘TX IPv4’)  
- ApiFrameType.REGISTER_JOINING_DEVICE = (36, ‘Register Joining Device’)  
- ApiFrameType.SEND_DATA_REQUEST = (40, ‘Send Data Request’)  
- ApiFrameType.DEVICE_RESPONSE = (42, ‘Device Response’)  
- ApiFrameType.USER_DATA_RELAY_REQUEST = (45, ‘User Data Relay Request’)  
- ApiFrameType.SOCKET_CREATE = (64, ‘Socket Create’)  
- ApiFrameType.SOCKET_OPTION_REQUEST = (65, ‘Socket Option Request’)  

---  

2.5. API reference
ApiFrameType.SOCKET_CONNECT = (66, ‘Socket Connect’)
ApiFrameType.SOCKET_CLOSE = (67, ‘Socket Close’)
ApiFrameType.SOCKET_SEND = (68, ‘Socket Send (Transmit)’)
ApiFrameType.SOCKET_SENDTO = (69, ‘Socket SendTo (Transmit Explicit Data): IPv4’)
ApiFrameType.SOCKET_BIND = (70, ‘Socket Bind/Listen’)
ApiFrameType.RX_64 = (128, ‘RX (Receive) Packet 64-bit Address’)
ApiFrameType.RX_16 = (129, ‘RX (Receive) Packet 16-bit Address’)
ApiFrameType.RX_IO_64 = (130, ‘IO Data Sample RX 64-bit Address Indicator’)
ApiFrameType.RX_IO_16 = (131, ‘IO Data Sample RX 16-bit Address Indicator’)
ApiFrameType.REMOTE_AT_COMMAND_RESPONSE_WIFI = (135, ‘Remote AT Command Response (Wi-Fi)’)
ApiFrameType.AT_COMMAND_RESPONSE = (136, ‘AT Command Response’)
ApiFrameType.TX_STATUS = (137, ‘TX (Transmit) Status’)
ApiFrameType.MODEM_STATUS = (138, ‘Modem Status’)
ApiFrameType.TRANSMIT_STATUS = (139, ‘Transmit Status’)
ApiFrameType.DIGIMESH_ROUTE_INFORMATION = (141, ‘Route Information’)
ApiFrameType.IO_DATA_SAMPLE_RX_INDICATOR_WIFI = (143, ‘IO Data Sample RX Indicator (Wi-Fi)’)
ApiFrameType.RECEIVE_PACKET = (144, ‘Receive Packet’)
ApiFrameType.EXPLICIT_RX_INDICATOR = (145, ‘Explicit RX Indicator’)
ApiFrameType.IO_DATA_SAMPLE_RX_INDICATOR = (146, ‘IO Data Sample RX Indicator’)
ApiFrameType.REMOTE_AT_COMMAND_RESPONSE = (151, ‘Remote Command Response’)
ApiFrameType.RX_SMS = (159, ‘RX SMS’)
ApiFrameType.ROUTE_RECORD_INDICATOR = (161, ‘Route Record Indicator’)
ApiFrameType.REGISTER_JOINING_DEVICE_STATUS = (164, ‘Register Joining Device Status’)
ApiFrameType.USER_DATA_RELAY_OUTPUT = (173, ‘User Data Relay Output’)
ApiFrameType.RX_IPV4 = (176, ‘RX IPv4’)
ApiFrameType.SEND_DATA_RESPONSE = (184, ‘Send Data Response’)
ApiFrameType.DEVICE_REQUEST = (185, ‘Device Request’)
ApiFrameType.DEVICE_RESPONSE_STATUS = (186, ‘Device Response Status’)
ApiFrameType.SOCKE Create_RESPONSE = (192, ‘Socket Create Response’)
ApiFrameType.SOCKE Option RESPONSE = (193, ‘Socket Option Response’)
ApiFrameType.SOCKE Connect RESPONSE = (194, ‘Socket Connect Response’)
ApiFrameType.SOCKE Close RESPONSE = (195, ‘Socket Close Response’)
ApiFrameType.SOCKE Listen RESPONSE = (198, ‘Socket Listen Response’)
ApiFrameType.SOCKE New_IPV4_CLIENT = (204, ‘Socket New IPv4 Client’)
ApiFrameType.SOCKE Receive = (205, ‘Socket Receive’)
ApiFrameType.SOCKE Receive FROM = (206, ‘Socket Receive From’)
ApiFrameType.SOCKE State = (207, ‘Socket State’)
ApiFrameType.FRAME_ERROR = (254, ‘Frame Error’)
ApiFrameType.GENERIC = (255, ‘Generic’)
ApiFrameType.UNKNOWN = (-1, ‘Unknown Packet’)

code
Integer. The API frame type code.
**description**
String. The API frame type description.

**digi.xbee.packets.base module**

class digi.xbee.packets.base.DictKeys
   Bases: enum.Enum
   
   This enumeration contains all keys used in dictionaries returned by to_dict() method of XBeePacket.

class digi.xbee.packets.base.XBeePacket
   Bases: object
   
   This abstract class represents the basic structure of an XBee packet.
   Derived classes should implement their own payload generation depending on their type.
   Generic actions like checksum compute or packet length calculation is performed here.
   Class constructor. Instantiates a new XBeePacket object.

def get_checksum()
   Returns the checksum value of this XBeePacket.
   
   The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
   
   Returns checksum value of this XBeePacket.
   
   Return type Integer

   See also:

   factory

output (escaped=False)
   Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.
   
   Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
   
   Returns raw bytearray of the XBeePacket.
   
   Return type Bytearray

def to_dict()
   Returns a dictionary with all information of the XBeePacket fields.
   
   Returns dictionary with all information of the XBeePacket fields.
   
   Return type Dictionary

static create_packet (raw, operating_mode)
   Abstract method. Creates a full XBeePacket with the given parameters. This function ensures that the
   XBeePacket returned is valid and is well built (if not exceptions are raised).
   
   If _OPERATING_MODE is API2 (API escaped) this method des-escape ‘raw’ and build the XBeePacket.
   Then, you can use XBeePacket.output() to get the escaped bytearray or not escaped.

   Parameters

   * raw (Bytearray) – bytearray with which the frame will be built. Must be a full frame
     represented by a bytearray.
• **operating_mode** (*OperatingMode*) – The mode in which the frame (‘byteArray’) was captured.

  **Returns** the XBee packet created.

  **Return type** *XBeePacket*

  **Raises** *InvalidPacketException* – if something is wrong with raw and the packet cannot be built well.

**get_frame_spec_data()**

Returns the data between the length field and the checksum field as bytearray. This data is never escaped.

  **Returns** the data between the length field and the checksum field as bytearray.

  **Return type** *Bytearray*

  **See also:**

```
factory
```

**static unescape_data** (*data*)

Un-escapes the provided bytearray data.

  **Parameters** *data* (*Bytearray*) – the bytearray to unescape.

  **Returns** *data* unescaped.

  **Return type** *Bytearray*

**class** *digi.xbee.packets.base.XBeeAPIPacket* (*api_frame_type*)

  **Bases:** *digi.xbee.packets.base.XBeePacket*

This abstract class provides the basic structure of a API frame.

Derived classes should implement their own methods to generate the API data and frame ID in case they support it.

Basic operations such as frame type retrieval are performed in this class.

**See also:**

```
XBeePacket
```

Class constructor. Instantiates a new *XBeeAPIPacket* object with the provided parameters.

  **Parameters** *api_frame_type* (*ApiFrameType* or Integer) – The API frame type.

  **See also:**

```
ApiFrameType
XBeePacket
```

**get_frame_spec_data()**

Override method.

  **See also:**
XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

needs_id()
Returns whether the packet requires frame ID or not.

Returns True if the packet needs frame ID, False otherwise.

Return type Boolean

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

static create_packet(raw, operating_mode)
Abstract method. Creates a full XBeePacket with the given parameters. This function ensures that the
XBeePacket returned is valid and is well built (if not exceptions are raised).

If _OPERATING_MODE is API2 (API escaped) this method des-escape ‘raw’ and build the XBeePacket.
Then, you can use XBeePacket.output() to get the escaped bytarray or not escaped.

Parameters

* raw (Bytearray) – bytarray with which the frame will be built. Must be a full frame
  represented by a bytarray.
• **operating_mode** (*OperatingMode*) – The mode in which the frame (‘byteArray’) was captured.

  Returns the XBe packet created.

  Return type **XBeePacket**

  Raises **InvalidPacketException** – if something is wrong with *raw* and the packet cannot be built well.

**get_checksum()**

  Returns the checksum value of this XBeePacket.

  The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

  Returns checksum value of this XBeePacket.

  Return type **Integer**

  See also:

  *factory*

**output** (*escaped=False*)

  Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

  Parameters **escaped** (*Boolean*) – indicates if the raw bytearray will be escaped or not.

  Returns raw bytearray of the XBeePacket.

  Return type **Bytearray**

**to_dict()**

  Returns a dictionary with all information of the XBeePacket fields.

  Returns dictionary with all information of the XBeePacket fields.

  Return type **Dictionary**

**static unescape_data** (*data*)

  Un-escapes the provided bytearray data.

  Parameters **data** (*Bytearray*) – the bytearray to unescape.

  Returns data unescaped.

  Return type **Bytearray**

**class** *digi.xbee.packets.base.GenericXBeePacket* (*rf_data*)

  Bases: *digi.xbee.packets.base.XBeeAPIPacket*

  This class represents a basic and Generic XBe packet.

  See also:

  *XBeeAPIPacket*

  Class constructor. Instantiates a **GenericXBeePacket** object with the provided parameters.

  Parameters **rf_data** (*bytearray*) – the frame specific data without frame type and frame ID.
static create_packet (raw, operating_mode=<OperatingMode.API_MODE: (1, 'API mode')>)

Override method.

Returns the GenericXBeePacket generated.

Return type GenericXBeePacket

Raises

- InvalidPacketException – if the bytearray length is less than 5. (start delim. + length (2 bytes) + frame type + checksum = 5 bytes).
- InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).
- InvalidPacketException – if the frame type is different than ApiFrameType.GENERIC.
- InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum ()

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.
Return type  Integer

See also:

factory

get_frame_spec_data()  
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()  
Returns the frame type of this packet.

Returns  the frame type of this packet.

Return type  ApiFrameType

See also:

ApiFrameType

get_frame_type_value()  
Returns the frame type integer value of this packet.

Returns  the frame type integer value of this packet.

Return type  Integer

See also:

ApiFrameType

is_broadcast()  
Returns whether this packet is broadcast or not.

Returns  True if this packet is broadcast, False otherwise.

Return type  Boolean

output (escaped=False)  
Returns the raw bytearray of thisXBeePacket, ready to be send by the serial port.

Parameters  escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns  raw bytearray of the XBeePacket.

Return type  Bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.
Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

```python
static unescape_data(data)
```

Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

```python
class digi.xbee.packets.base.UnknownXBeePacket(api_frame, rf_data)
```

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an unknown XBee packet.

See also:

```
XBeeAPIPacket
```

Class constructor. Instantiates a UnknownXBeePacket object with the provided parameters.

Parameters

- **api_frame** (Integer) – the API frame integer value of this packet.
- **rf_data** (bytearray) – the frame specific data without frame type and frame ID.

See also:

```
factory
XBeeAPIPacket
```

```python
static create_packet(raw, operating_mode=<OperatingMode.API_MODE: (1, 'API mode')>)
```

Override method.

Returns the UnknownXBeePacket generated.

Return type UnknownXBeePacket

Raises

- **InvalidPacketException** – if the bytearray length is less than 5. (start delim. + length (2 bytes) + frame type + checksum = 5 bytes).
- **InvalidPacketException** – if the length field of `raw` is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of `raw` is not the header byte. See `SpecialByte`.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType
is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.
Return type Boolean

needs_id()
Override method.
See also:

XBeeAPacket.needs_id()

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.
Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.
Return type Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.
Return type Bytearray

digi.xbee.packets.cellular module
digi.xbee.packets.cellular.PATTERN_PHONE_NUMBER = '^\+?\d+$'
Pattern used to validate the phone number parameter of SMS packets.

class digi.xbee.packets.cellular.RXSMSPacket (phone_number, data)
Bases: digi.xbee.packets.base.XBeeAPacket

This class represents an RX (Receive) SMS packet. Packet is built using the parameters of the constructor or providing a valid byte array.
See also:

TXSMSPacket
XBeeAPacket

Class constructor. Instantiates a new RXSMSPacket object with the provided parameters.
Parameters
• **phone_number** (*String*) – phone number of the device that sent the SMS.
• **data** (*String*) – packet data (text of the SMS).

**Raises**
• **ValueError** – if length of **phone_number** is greater than 20.
• **ValueError** – if **phone_number** is not a valid phone number.

**static create_packet** (*raw, operating_mode*)
Override method.

**Returns** **RXSMSPacket**

**Raises**
• **InvalidPacketException** – if the bytearray length is less than 25. (start delim + length (2 bytes) + frame type + phone number (20 bytes) + checksum = 25 bytes)
• **InvalidPacketException** – if the length field of **raw** is different than its real length. (length field: bytes 2 and 3)
• **InvalidPacketException** – if the first byte of **raw** is not the header byte. See **SPECIAL_BYTE**.
• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
• **InvalidPacketException** – if the frame type is different than **ApiFrameType. RX_SMS**.
• **InvalidOperatingModeException** – if **operating_mode** is not supported.

See also:

`XBeePacket.create_packet()`

`needs_id()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`get_phone_number_byte_array()`
Returns the phone number byte array.

**Returns** phone number of the device that sent the SMS.

**Return type** Bytearray

`phone_number`
String. Phone number that sent the SMS.

`data`
String. Data of the SMS.

`frame_id`
Returns the frame ID of the packet.
**Returns** the frame ID of the packet.

**Return type** Integer

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

**Returns** checksum value of this XBeePacket.

**Return type** Integer

See also:

*factory*

**get_frame_spec_data()**

Override method.

See also:

*XBeePacket.get_frame_spec_data()*

**get_frame_type()**

Returns the frame type of this packet.

**Returns** the frame type of this packet.

**Return type** ApiFrameType

See also:

*ApiFrameType*

**get_frame_type_value()**

Returns the frame type integer value of this packet.

**Returns** the frame type integer value of this packet.

**Return type** Integer

See also:

*ApiFrameType*

**is_broadcast()**

Returns whether this packet is broadcast or not.

**Returns** True if this packet is broadcast, False otherwise.

**Return type** Boolean
output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters
escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.
Return type bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.
Returns dictionary with all information of the XBeePacket fields.
Return type Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.
Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.
Return type bytearray

class digi.xbee.packets.cellular.TXSMSPacket (frame_id, phone_number, data)
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a TX (Transmit) SMS packet. Packet is built using the parameters of the constructor or providing a valid byte array.

See also:

RXSMSPacket
XBeeAPIPacket

Class constructor. Instantiates a new TXSMSPacket object with the provided parameters.

Parameters
frame_id (Integer) – the frame ID. Must be between 0 and 255.
phone_number (String) – the phone number.
data (String) – this packet’s data.

Raises
ValueError – if frame_id is not between 0 and 255.
ValueError – if length of phone_number is greater than 20.
ValueError – if phone_number is not a valid phone number.

See also:

XBeeAPIPacket

static create_packet (raw, operating_mode)
Override method.
Returns `TXSMSPacket`

Raises

- `InvalidPacketException` – if the bytearray length is less than 27. (start delim, length (2 bytes), frame type, frame id, transmit options, phone number (20 bytes), checksum)

- `InvalidPacketException` – if the length field of `raw` is different than its real length. (length field: bytes 2 and 3)

- `InvalidPacketException` – if the first byte of `raw` is not the header byte. See `SPECIAL_BYTE`.

- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).

- `InvalidPacketException` – if the frame type is different than `ApiFrameType.TX_SMS`.

- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

`XBeePacket.create_packet()`

`needs_id()`

Override method.

See also:

`XBeeAPIPacket.needs_id()`

`get_phone_number_byte_array()`

Returns the phone number byte array.

Returns phone number of the device that sent the SMS.

Return type Bytearray

`frame_id`

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

`get_checksum()`

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:
factory

get_frame_spec_data()
Override method.
See also:
XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.
   Returns the frame type of this packet.
   Return type ApiFrameType
See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.
   Returns the frame type integer value of this packet.
   Return type Integer
See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.
   Returns True if this packet is broadcast, False otherwise.
   Return type Boolean

output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.
   Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
   Returns raw bytearray of the XBeePacket.
   Return type bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.
   Returns dictionary with all information of the XBeePacket fields.
   Return type Dictionary
static unescape_data(data)
    Un-escapes the provided bytearray data.
    
    Parameters data (Bytearray) – the bytearray to unescape.
    Returns data unescaped.
    Return type Bytearray

phone_number
    String. Phone number that sent the SMS.

data
    String. Data of the SMS.

**digi.xbee.packets.common module**

class digi.xbee.packets.common.ATCommPacket (frame_id, command, parameter=None)
    Bases: digi.xbee.packets.base.XBeeAPIPacket

    This class represents an AT command packet.
    Used to query or set module parameters on the local device. This API command applies changes after executing
    the command. (Changes made to module parameters take effect once changes are applied.).
    Command response is received as an ATCommResponsePacket.
    See also:

    ATCommResponsePacket
    XBeeAPIPacket

    Class constructor. Instantiates a new ATCommPacket object with the provided parameters.
    
    Parameters
    
    • frame_id (Integer) – the frame ID of the packet.
    • command (String) – the AT command of the packet. Must be a string.
    • parameter (Bytearray, optional) – the AT command parameter. Optional.
    
    Raises
    
    • ValueError – if frame_id is less than 0 or greater than 255.
    • ValueError – if length of command is different than 2.
    
    See also:

    XBeeAPIPacket

static create_packet (raw, operating_mode)
    Override method.
    
    Returns ATCommPacket
    Raises
• **InvalidPacketException** – if the bytearray length is less than 6. (start delim. + length (2 bytes) + frame type + frame id + checksum = 6 bytes).

• **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.

• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).

• **InvalidPacketException** – if the frame type is different than `ApiFrameType`. `AT_COMMAND`.

• **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

```python
needs_id()
```

Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

**command**

String. AT command.

**parameter**

Bytearray. AT command parameter.

**frame_id**

Returns the frame ID of the packet.

- **Returns** the frame ID of the packet.

- **Return type** Integer

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

- **Returns** checksum value of this XBeePacket.

- **Return type** Integer

See also:

```python
factory
```
get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type bytearray

to_dict()

Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)

Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.

Return type Bytearray
class digi.xbee.packets.common.ATCommQueuePacket (frame_id, command, parameter=None)
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an AT command Queue packet.

Used to query or set module parameters on the local device.

In contrast to the ATCommPacket API packet, new parameter values are queued and not applied until either an ATCommPacket is sent or the applyChanges() method of the XBeeDevice class is issued.

Command response is received as an ATCommResponsePacket.

See also:

ATCommResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new ATCommQueuePacket object with the provided parameters.

Parameters

- frame_id (Integer) – the frame ID of the packet.
- command (String) – the AT command of the packet. Must be a string.
- parameter (Bytearray, optional) – the AT command parameter. Optional.

Raises

- ValueError – if frame_id is less than 0 or greater than 255.
- ValueError – if length of command is different than 2.

See also:

XBeeAPIPacket

static create_packet (raw, operating_mode)
Override method.

Returns ATCommQueuePacket

Raises

- InvalidPacketException – if the bytarray length is less than 6. (start delim. + length (2 bytes) + frame type + frame id + checksum = 6 bytes).
- InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).
• **InvalidPacketException** – if the frame type is different than `ApiFrameType.AT_COMMAND_QUEUE`.

• **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

```python
needs_id()
```

Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

```python
command
```

String. AT command.

```python
parameter
```

Bytestring. AT command parameter.

```python
frame_id
```

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type: Integer

```python
get_checksum()
```

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type: Integer

See also:

```python
factory
```

```python
get_frame_spec_data()
```

Override method.

See also:

```python
XBeePacket.get_frame_spec_data()
```

```python
get_frame_type()
```

Returns the frame type of this packet.
Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output (escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type bytearray

to_dict()

Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data (data)

Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type bytearray

class digi.xbee.packets.common.ATCommResponsePacket (frame_id, command, response_status=<ATCommandStatus.OK: (0, 'Status OK')>, comm_value=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an AT command response packet.

In response to an AT command message, the module will send an AT command response message. Some commands will send back multiple frames (for example, the ND - Node Discover command).
This packet is received in response of an `ATCommPacket`.
Response also includes an `ATCommandStatus` object with the status of the AT command.

See also:

`ATCommPacket`  
`ATCommandStatus`  
`XBeeAPIPacket`

Class constructor. Instantiates a new `ATCommResponsePacket` object with the provided parameters.

**Parameters**

- `frame_id` (**Integer**) – the frame ID of the packet. Must be between 0 and 255.
- `command` (**String**) – the AT command of the packet. Must be a string.
- `response_status` (**ATCommandStatus** or **Integer**) – the status of the AT command.

**Raises**

- `ValueError` – if `frame_id` is less than 0 or greater than 255.
- `ValueError` – if length of `command` is different than 2.

See also:

`ATCommandStatus`  
`XBeeAPIPacket`

**static create_packet (raw, operating_mode)**

Override method.

**Returns** `ATCommResponsePacket`

**Raises**

- `InvalidPacketException` – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + frame id + at command (2 bytes) + command status + checksum = 9 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different than `ApiFrameType.AT_COMMAND_RESPONSE`.
- `InvalidPacketException` – if the command status field is not a valid value. See `ATCommandStatus`.  

---

**2.5. API reference**

169
• `InvalidOperatingModeException` – if operating_mode is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

`needs_id()`  
Override method.  

See also:

```python
XBeeAPIPacket.needs_id()
```

`command`

String. AT command.

`command_value`

Bytearray. AT command value.

`status`

`ATCommandStatus`. AT command response status.

`real_status`

Integer. AT command response status.

`frame_id`

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type  Integer

`get_checksum()`

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type  Integer

See also:

```python
factory
```

`get_frame_spec_data()`  
Override method.  

See also:

```python
XBeePacket.get_frame_spec_data()
```
get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type bytearray

class digi.xbee.packets.common.ReceivePacket(x64bit_addr, x16bit_addr, receive_options, rf_data=None)
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a receive packet. Packet is built using the parameters of the constructor or providing a valid byte array.
When the module receives an RF packet, it is sent out the UART using this message type.

This packet is received when external devices send transmit request packets to this module.

Among received data, some options can also be received indicating transmission parameters.

See also:

TransmitPacket
ReceiveOptions
XBeeAPIPacket

Class constructor. Instantiates a new ReceivePacket object with the provided parameters.

Parameters

- `x64bit_addr` (XBee64BitAddress) – the 64-bit source address.
- `x16bit_addr` (XBee16BitAddress) – the 16-bit source address.
- `receive_options` (Integer) – bitfield indicating the receive options.
- `rf_data` (Bytearray, optional) – received RF data. Optional.

See also:

ReceiveOptions
XBee16BitAddress
XBee64BitAddress
XBeeAPIPacket

static `create_packet` (raw, operating_mode)

Override method.

Returns ATCommResponsePacket

Raises

- `InvalidPacketException` – if the bytearray length is less than 16. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + 16bit addr. + Receive options + checksum = 16 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not ApiFrameType.RECEIVE_PACKET.
- `InvalidOperatingModeException` – if operating_mode is not supported.

See also:
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

is_broadcast()
Override method.

See also:

XBeeAPIPacket.is_broadcast()

x64bit_source_addr
XBee64BitAddress. 64-bit source address.

x16bit_source_addr
XBee16BitAddress. 16-bit source address.

receive_options
Integer. Receive options bitfield.

rf_data
Bytarray. Received RF data.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:
get_frame_type()  
Returns the frame type of this packet.

    Returns the frame type of this packet.
    Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()  
Returns the frame type integer value of this packet.

    Returns the frame type integer value of this packet.
    Return type Integer

See also:

ApiFrameType

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

    Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
    Returns raw bytearray of the XBeePacket.
    Return type Bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.

    Returns dictionary with all information of the XBeePacket fields.
    Return type Dictionary

static unescape_data(data)  
Un-escapes the provided bytearray data.

    Parameters data (Bytearray) – the bytearray to unescape.
    Returns data unescaped.
    Return type Bytearray

class digi.xbee.packets.common.RemoteATCommandPacket(frame_id, x64bit_addr, x16bit_addr, transmit_options, command, parameter=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Remote AT command Request packet. Packet is built using the parameters of the constructor or providing a valid byte array.
Used to query or set module parameters on a remote device. For parameter changes on the remote device to take effect, changes must be applied, either by setting the apply changes options bit, or by sending an AC command to the remote node.

Remote command options are set as a bitfield.

If configured, command response is received as a `RemoteATCommandResponsePacket`.

See also:

`RemoteATCommandResponsePacket`
`XBeeAPIPacket`

Class constructor. Instantiates a new `RemoteATCommandPacket` object with the provided parameters.

**Parameters**

- `frame_id (integer)` – the frame ID of the packet.
- `x64bit_addr (XBee64BitAddress)` – the 64-bit destination address.
- `x16bit_addr (XBee16BitAddress)` – the 16-bit destination address.
- `transmit_options (Integer)` – bitfield of supported transmission options.
- `command (String)` – AT command to send.
- `parameter (Bytearray, optional)` – AT command parameter. Optional.

**Raises**

- `ValueError` – if `frame_id` is less than 0 or greater than 255.
- `ValueError` – if length of `command` is different than 2.

See also:

`RemoteATCmdOptions`
`XBee16BitAddress`
`XBee64BitAddress`
`XBeeAPIPacket`

**static create_packet (raw, operating_mode)**

Override method.

**Returns** `RemoteATCommandPacket`

**Raises**

- `InvalidPacketException` – if the Bytearray length is less than 19. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + 16bit addr. + transmit options + command (2 bytes) + checksum = 19 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`. 
• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).

• **InvalidPacketException** – if the frame type is not ApiFrameType. REMOTE_AT_COMMAND_REQUEST.

• **InvalidOperatingModeException** – if operating_mode is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

**needs_id()**

Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

**x64bit_dest_addr**

`XBee64BitAddress`. 64-bit destination address.

**x16bit_dest_addr**

`XBee16BitAddress`. 16-bit destination address.

**transmit_options**

Integer. Transmit options bitfield.

**command**

String. AT command.

**parameter**

ByteArray. AT command parameter.

**frame_id**

Returns the frame ID of the packet.

- **Returns** the frame ID of the packet.

- **Return type** Integer

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

- **Returns** checksum value of this XBeePacket.

- **Return type** Integer

See also:

```python
factory
```
get_frame_spec_data()  
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()  
Returns the frame type of this packet.

    Returns the frame type of this packet.
    Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()  
Returns the frame type integer value of this packet.

    Returns the frame type integer value of this packet.
    Return type Integer

See also:

ApiFrameType

is_broadcast()  
Returns whether this packet is broadcast or not.

    Returns True if this packet is broadcast, False otherwise.
    Return type Boolean

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

    Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
    Returns raw bytearray of the XBeePacket.
    Return type bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.

    Returns dictionary with all information of the XBeePacket fields.
    Return type Dictionary

static unescape_data(data)  
Un-escapes the provided bytearray data.

    Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.common.RemoteATCommandResponsePacket (frame_id, x64bit_addr, x16bit_addr, command, response_status, comm_value=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a remote AT command response packet. Packet is built using the parameters of the constructor or providing a valid byte array.

If a module receives a remote command response RF data frame in response to a remote AT command request, the module will send a remote AT command response message out the UART. Some commands may send back multiple frames, for example, Node Discover (ND) command.

This packet is received in response of a RemoteATCommandPacket.

Response also includes an object with the status of the AT command.

See also:

RemoteATCommandPacket
ATCommandStatus
XBeeAPIPacket

Class constructor. Instantiates a new RemoteATCommandResponsePacket object with the provided parameters.

Parameters

• frame_id (Integer) – the frame ID of the packet.
• x64bit_addr (XBee64BitAddress) – the 64-bit source address
• x16bit_addr (XBee16BitAddress) – the 16-bit source address.
• command (String) – the AT command of the packet. Must be a string.
• response_status (ATCommandStatus or Integer) – the status of the AT command.
• comm_value (Bytearray, optional) – the AT command response value. Optional.

Raises

• ValueError – if frame_id is less than 0 or greater than 255.
• ValueError – if length of command is different than 2.

See also:

ATCommandStatus
XBee16BitAddress
XBee64BitAddress
XBeeAPIPacket
static create_packet (raw, operating_mode)
Override method.

Returns RemoteATCommandResponsePacket.

Raises

- InvalidPacketException – if the bytearray length is less than 19. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + 16bit addr. + receive options + command (2 bytes) + checksum = 19 bytes).
- InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).
- InvalidPacketException – if the frame type is not ApiFrameType.REMOTE_AT_COMMAND_RESPONSE.
- InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

x64bit_source_addr
XBee64BitAddress. 64-bit source address.

x16bit_source_addr
XBee16BitAddress. 16-bit source address.

command
String. AT command.

command_value
Bytearray. AT command value.

status
ATCommandStatus. AT command response status.

real_status
Integer. AT command response status.

frame_id
Returns the frame ID of the packet.
Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean
output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.
Return type Bytearray

to_dict ()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.
Return type Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.
Return type Bytearray

class digi.xbee.packets.common.TransmitPacket (frame_id, x64bit_addr, x16bit_addr, broadcast_radius, transmit_options, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a transmit request packet. Packet is built using the parameters of the constructor or providing a valid API byte array.

A transmit request API frame causes the module to send data as an RF packet to the specified destination.

The 64-bit destination address should be set to 0x000000000000FFFF for a broadcast transmission (to all devices).

The coordinator can be addressed by either setting the 64-bit address to all 0x00} and the 16-bit address to 0xFFFE, OR by setting the 64-bit address to the coordinator’s 64-bit address and the 16-bit address to 0x0000.

For all other transmissions, setting the 16-bit address to the correct 16-bit address can help improve performance when transmitting to multiple destinations.

If a 16-bit address is not known, this field should be set to 0xFFFE (unknown).

The transmit status frame (ApiFrameType.TRANSMIT_STATUS) will indicate the discovered 16-bit address, if successful (see TransmitStatusPacket).

The broadcast radius can be set from 0 up to NH. If set to 0, the value of NH specifies the broadcast radius (recommended). This parameter is only used for broadcast transmissions.

The maximum number of payload bytes can be read with the NP command.

Several transmit options can be set using the transmit options bitfield.

See also:

TransmitOptions
XBeel6BitAddress.COORDINATOR_ADDRESS
XBeel6BitAddress.UNKNOWN_ADDRESS
XBeel64BitAddress.BROADCAST_ADDRESS
XBeel64BitAddress.COORDINATOR_ADDRESS

2.5. API reference
Class constructor. Instantiates a new TransmitPacket object with the provided parameters.

**Parameters**

- `frame_id` (*integer*) – the frame ID of the packet.
- `x64bit_addr` (*XBee64BitAddress*) – the 64-bit destination address.
- `x16bit_addr` (*XBee16BitAddress*) – the 16-bit destination address.
- `broadcast_radius` (*Integer*) – maximum number of hops a broadcast transmission can occur.
- `transmit_options` (*Integer*) – bitfield of supported transmission options.
- `rf_data` (*Bytearray, optional*) – RF data that is sent to the destination device. Optional.

See also:

**Raises** `ValueError` – if `frame_id` is less than 0 or greater than 255.

**static create_packet** (*raw, operating_mode*)

Override method.

Returns `TransmitPacket`.

Raises

- `InvalidPacketException` – if the bytearray length is less than 18. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + 16bit addr. + Receive options + checksum = 16 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.TRANSMIT_REQUEST`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

`XBeePacket.create_packet()`  
`XBeeAPIPacket._check_api_packet()`
needs_id()
   Override method.

See also:

XBeeAPIPacket.needs_id()

x64bit_dest_addr
   XBee64BitAddress. 64-bit destination address.

x16bit_dest_addr
   XBee16BitAddress. 16-bit destination address.

transmit_options
   Integer. Transmit options bitfield.

broadcast_radius
   Integer. Broadcast radius.

rf_data
   Bytearray. RF data to send.

frame_id
   Returns the frame ID of the packet.
   
   Returns the frame ID of the packet.
   
   Return type Integer

get_checksum()
   Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

   Returns checksum value of this XBeePacket.
   
   Return type Integer

See also:

factory

get_frame_spec_data()
   Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
   Returns the frame type of this packet.

   Returns the frame type of this packet.
   
   Return type ApiFrameType
See also:

 ApiServiceType

**get_frame_type_value()**
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

**is_broadcast()**
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

**output**(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped(Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

**to_dict()**
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

**static unescape_data**(data)
Un-escapes the provided bytearray data.

Parameters data(Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

**class** di.xbee.packets.common.TransmitStatusPacket(frame_id, x16bit_addr, transmit_retry_count, transmit_status=<TransmitStatus.SUCCESS: 0, 'Success.'>, discovery_status=<DiscoveryStatus.NO_DISCOVERY_OVERHEAD: 0, 'No discovery overhead'>)

Bases: di.xbee.packets.base.XBeeAPIPacket

This class represents a transmit status packet. Packet is built using the parameters of the constructor or providing a valid raw byte array.

When a Transmit Request is completed, the module sends a transmit status message. This message will indicate if the packet was transmitted successfully or if there was a failure.
This packet is the response to standard and explicit transmit requests.

See also:

TransmitPacket

Class constructor. Instantiates a new TransmitStatusPacket object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **x16bit_addr** (*XBee16BitAddress*) – 16-bit network address the packet was delivered to.
- **transmit_retry_count** (*Integer*) – the number of application transmission retries that took place.

Raises **ValueError** – if frame_id is less than 0 or greater than 255.

See also:

DiscoveryStatus
TransmitStatus
XBee16BitAddress
XBeeAPIPacket

**static create_packet** (*raw, operating_mode*)
Override method.

Returns **TransmitStatusPacket**

Raises

- **InvalidPacketException** – if the bytearray length is less than 11. (start delim. + length (2 bytes) + frame type + frame id + 16bit addr. + transmit retry count + delivery status + discovery status + checksum = 11 bytes).
- **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is not ApiFrameType.TRANSMIT_STATUS.
- **InvalidOperatingModeException** – if operating_mode is not supported.
See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.
See also:

XBeeAPIPacket.needs_id()

16bit_dest_addr
XBee16BitAddress. 16-bit destination address.

transmit_retry_count
Integer. Transmit retry count value.

transmit_status
TransmitStatus. Transmit status.

discovery_status
DiscoveryStatus. Discovery status.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

generic_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer
See also:

factory

generic_frame_spec_data()
Override method.
See also:

XBeePacket.get_frame_spec_data()
get_frame_type()

Returns the frame type of this packet.

**Returns**

the frame type of this packet.

**Return type**

ApiFrameType

See also:

 ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

**Returns**

the frame type integer value of this packet.

**Return type**

Integer

See also:

 ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

**Returns**

True if this packet is broadcast, False otherwise.

**Return type**

Boolean

output (escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

**Parameters**

escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

**Returns**

raw bytearray of the XBeePacket.

**Return type**

Bytearray
to_dict()

Returns a dictionary with all information of the XBeePacket fields.

**Returns**

dictionary with all information of the XBeePacket fields.

**Return type**

Dictionary

static unescape_data (data)

Un-escapes the provided bytearray data.

**Parameters**

data (Bytearray) – the bytearray to unescape.

**Returns**

data unescaped.

**Return type**

Bytearray
class digi.xbee.packets.common.ModemStatusPacket (modem_status)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a modem status packet. Packet is built using the parameters of the constructor or providing a valid API raw byte array.
RF module status messages are sent from the module in response to specific conditions and indicates the state of the modem in that moment.

See also:

**XBeeAPIPacket**

Class constructor. Instantiates a new `ModemStatusPacket` object with the provided parameters.

**Parameters** `modem_status` (*ModemStatus*) – the modem status event.

See also:

**ModemStatus**

**XBeeAPIPacket**

**static create_packet** (*raw, operating_mode*)

Override method.

**Returns** `ModemStatusPacket`.

**Raises**

- `InvalidPacketException` – if the bytearray length is less than 6. (start delim. + length (2 bytes) + frame type + modem status + checksum = 6 bytes).
- `InvalidPacketException` – if the length field of `raw` is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of `raw` is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.MODEM_STATUS`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

**XBeePacket.create_packet()**

**XBeeAPIPacket._check_api_packet()**

**needs_id()**

Override method.

See also:

**XBeeAPIPacket.needs_id()**
modem_status

ModemStatus. Modem status event.

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType
**is_broadcast()**

Returns whether this packet is broadcast or not.

- **Returns**: True if this packet is broadcast, False otherwise.
- **Return type**: Boolean

**output (escaped=False)**

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

- **Parameters** escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
- **Returns**: raw bytearray of the XBeePacket.
- **Return type**: Bytearray

**to_dict()**

Returns a dictionary with all information of the XBeePacket fields.

- **Returns**: dictionary with all information of the XBeePacket fields.
- **Return type**: Dictionary

**static unescape_data (data)**

Un-escapes the provided bytearray data.

- **Parameters** data (Bytearray) – the bytearray to unescape.
- **Returns**: data unescaped.
- **Return type**: Bytearray

---

**class digi.xbee.packets.common.IODataSampleRxIndicatorPacket (x64bit_addr, x16bit_addr, receive_options, rf_data=None)**

**Bases**: digi.xbee.packets.base.XBeeAPIPacket

This class represents an IO data sample RX indicator packet. Packet is built using the parameters of the constructor or providing a valid API byte array.

When the module receives an IO sample frame from a remote device, it sends the sample out the UART using this frame type (when AO=0). Only modules running API firmware will send IO samples out the UART.

Among received data, some options can also be received indicating transmission parameters.

**See also:**

- XBeeAPIPacket
- ReceiveOptions

---

**Class constructor.** Instantiates a new IODataSampleRxIndicatorPacket object with the provided parameters.

**Parameters**

- **x64bit_addr (XBee64BitAddress)** – the 64-bit source address.
- **x16bit_addr (XBee16BitAddress)** – the 16-bit source address.
- **receive_options (Integer)** – bitfield indicating the receive options.
- **rf_data (Bytearray, optional)** – received RF data. Optional.
Raises `ValueError` – if `rf_data` is not `None` and it’s not valid for create an `IOSample`.

See also:

- `IOSample`
- `ReceiveOptions`
- `XBee16BitAddress`
- `XBee64BitAddress`
- `XBeeAPIPacket`

**static `create_packet`** *(raw, operating_mode)*

Override method.

Returns `IODataSampleRxIndicatorPacket`.

Raises

- `InvalidPacketException` – if the bytearray length is less than 20. (start delim. + length (2 bytes) + frame type + 64bit addr. + 16bit addr. + rf data (5 bytes) + checksum = 20 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.IO_DATA_SAMPLE_RX_INDICATOR`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

```
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

**needs_id()**

Override method.

See also:

```
XBeeAPIPacket.needs_id()
```

**is_broadcast()**

Override method.

See also:
XBeeAPIPacket.is_broadcast()

**x64bit_source_addr**
*XBee64BitAddress* 64-bit source address.

**x16bit_source_addr**
*XBee16BitAddress* 16-bit source address.

**receive_options**
Integer. Receive options bitfield.

**rf_data**
Bytarray. Received RF data.

**io_sample**
IO sample corresponding to the data contained in the packet.

  Type *IOSample*

**frame_id**
Returns the frame ID of the packet.

  Returns the frame ID of the packet.

  Return type Integer

**get_checksum()**
Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

  Returns checksum value of this XBeePacket.

  Return type Integer

See also:

*factory*

**get_frame_spec_data()**
Override method.

See also:

*XBeePacket.get_frame_spec_data()*

**get_frame_type()**
Returns the frame type of this packet.

  Returns the frame type of this packet.

  Return type *ApiFrameType*

See also:

*ApiFrameType*
get_frame_type_value()
Returns the frame type integer value of this packet.

- **Returns**: the frame type integer value of this packet.
- **Return type**: Integer

See also:

ApiFrameType

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

- **Parameters**: escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
- **Returns**: raw bytearray of the XBeePacket.
- **Return type**: Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

- **Returns**: dictionary with all information of the XBeePacket fields.
- **Return type**: Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.

- **Parameters**: data (Bytearray) – the bytearray to unescape.
- **Returns**: data unescaped.
- **Return type**: Bytearray

class digi.xbee.packets.common.ExplicitAddressingPacket (frame_id, x64bit_addr, x16bit_addr, source_endpoint, dest_endpoint, cluster_id, profile_id, broadcast_radius=0, transmit_options=0, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an explicit addressing command packet. Packet is built using the parameters of the constructor or providing a valid API payload.

Allows application layer fields (endpoint and cluster ID) to be specified for a data transmission. Similar to the transmit request, but also requires application layer addressing fields to be specified (endpoints, cluster ID, profile ID). An explicit addressing request API frame causes the module to send data as an RF packet to the specified destination, using the specified source and destination endpoints, cluster ID, and profile ID.

The 64-bit destination address should be set to 0x000000000000FFFF for a broadcast transmission (to all devices).

The coordinator can be addressed by either setting the 64-bit address to all 0x00 and the 16-bit address to 0xFFFF, OR by setting the 64-bit address to the coordinator’s 64-bit address and the 16-bit address to 0x0000.

For all other transmissions, setting the 16-bit address to the correct 16-bit address can help improve performance when transmitting to multiple destinations.

2.5. API reference
If a 16-bit address is not known, this field should be set to 0xFFFE (unknown).

The transmit status frame (ApiFrameType.TRANSMIT_STATUS) will indicate the discovered 16-bit address, if successful (see TransmitStatusPacket).

The broadcast radius can be set from 0 up to NH. If set to 0, the value of NH specifies the broadcast radius (recommended). This parameter is only used for broadcast transmissions.

The maximum number of payload bytes can be read with the NP command. Note: if source routing is used, the RF payload will be reduced by two bytes per intermediate hop in the source route.

Several transmit options can be set using the transmit options bitfield.

See also:

TransmitOptions
XBee16BitAddress.COORDINATOR_ADDRESS
XBee16BitAddress.UNKNOWN_ADDRESS
XBee64BitAddress.BROADCAST_ADDRESS
XBee64BitAddress.COORDINATOR_ADDRESS
ExplicitRXIndicatorPacket
XBeeAPIPacket

Class constructor. . Instantiates a new ExplicitAddressingPacket object with the provided parameters.

Parameters

- **frame_id** (Integer) – the frame ID of the packet.
- **x64bit_addr** (XBee64BitAddress) – the 64-bit address.
- **x16bit_addr** (XBee16BitAddress) – the 16-bit address.
- **source_endpoint** (Integer) – source endpoint. 1 byte.
- **dest_endpoint** (Integer) – destination endpoint. 1 byte.
- **cluster_id** (Integer) – cluster id. Must be between 0 and 0xFFFF.
- **profile_id** (Integer) – profile id. Must be between 0 and 0xFFFF.
- **broadcast_radius** (Integer) – maximum number of hops a broadcast transmission can occur.
- **transmit_options** (Integer) – bitfield of supported transmission options.
- **rf_data** (Bytearray, optional) – RF data that is sent to the destination device. Optional.

Raises

- **ValueError** – if frame_id, src_endpoint or dst_endpoint are less than 0 or greater than 255.
- **ValueError** – if lengths of cluster_id or profile_id (respectively) are less than 0 or greater than 0xFFFF.

See also:

XBee16BitAddress
static `create_packet` *(raw, operating_mode)*

Override method.

Returns `ExplicitAddressingPacket`.

Raises

- `InvalidPacketException` – if the bytearray length is less than 24. (start delim. + length (2 bytes) + frame type + frame ID + 64bit addr. + 16bit addr. + source endpoint + dest. endpoint + cluster ID (2 bytes) + profile ID (2 bytes) + broadcast radius + transmit options + checksum = 24 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different than `ApiFrameType.EXPLICIT_ADDRESSING`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

- `XBeePacket.create_packet()`
- `XBeeAPIPacket._check_api_packet()`

`needs_id()`

Override method.

See also:

- `XBeeAPIPacket.needs_id()`

`x64bit_dest_addr`

`XBe64BitAddress`. 64-bit destination address.

`x16bit_dest_addr`

`XBe16BitAddress`. 16-bit destination address.

`transmit_options`

Integer. Transmit options bitfield.

`broadcast_radius`

Integer. Broadcast radius.
frame_id
Returns the frame ID of the packet.
    Returns the frame ID of the packet.
    Return type Integer
get_checksum()
Returns the checksum value of this XBeePacket.
The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
    Returns checksum value of this XBeePacket.
    Return type Integer
See also:

factory

get_frame_spec_data()
Override method.
See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.
    Returns the frame type of this packet.
    Return type ApiFrameType
See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.
    Returns the frame type integer value of this packet.
    Return type Integer
See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.
    Returns True if this packet is broadcast, False otherwise.
Return type Boolean

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type bytearray

source_endpoint
Integer. Source endpoint of the transmission.

to_dict ()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type bytearray

dest_endpoint
Integer. Destination endpoint of the transmission.

cluster_id
Integer. Cluster ID of the transmission.

profile_id
Integer. Profile ID of the transmission.

rf_data
Bytearray. RF data to send.

class digi.xbee.packets.common.ExplicitRXIndicatorPacket (x64bit_addr, x16bit_addr, source_endpoint, dest_endpoint, cluster_id, profile_id, receive_options, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an explicit RX indicator packet. Packet is built using the parameters of the constructor or providing a valid API payload.

When the modem receives an RF packet it is sent out the UART using this message type (when AO=1).

This packet is received when external devices send explicit addressing packets to this module.

Among received data, some options can also be received indicating transmission parameters.

See also:

XBeeReceiveOptions
ExplicitAddressingPacket
XBeeAPIPacket

Class constructor. Instantiates a new ExplicitRXIndicatorPacket object with the provided parameters.

Parameters

- **x64bit_addr** (`XBee64BitAddress`) – the 64-bit source address.
- **x16bit_addr** (`XBee16BitAddress`) – the 16-bit source address.
- **source_endpoint** (`Integer`) – source endpoint. 1 byte.
- **dest_endpoint** (`Integer`) – destination endpoint. 1 byte.
- **cluster_id** (`Integer`) – cluster ID. Must be between 0 and 0xFFFF.
- **profile_id** (`Integer`) – profile ID. Must be between 0 and 0xFFFF.
- **receive_options** (`Integer`) – bitfield indicating the receive options.
- **rf_data** (`Bytearray, optional`) – received RF data. Optional.

Raises

- **ValueError** – if src_endpoint or dst_endpoint are less than 0 or greater than 255.
- **ValueError** – if lengths of cluster_id or profile_id (respectively) are different than 2.

See also:

- `XBee16BitAddress`
- `XBee64BitAddress`
- `XBeeReceiveOptions`
- `XBeeAPIPacket`

**frame_id**

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type `Integer`

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type `Integer`

See also:

- `factory`
get_frame_spec_data()
  Override method.

  See also:

  XBeePacket.get_frame_spec_data()

get_frame_type()
  Returns the frame type of this packet.

    Returns  the frame type of this packet.

    Return type  ApiFrameType

  See also:

  ApiFrameType

get_frame_type_value()
  Returns the frame type integer value of this packet.

    Returns  the frame type integer value of this packet.

    Return type  Integer

  See also:

  ApiFrameType

output(escaped=False)
  Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

    Parameters  escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

    Returns  raw bytearray of the XBeePacket.

    Return type  Bytearray

to_dict()
  Returns a dictionary with all information of the XBeePacket fields.

    Returns  dictionary with all information of the XBeePacket fields.

    Return type  Dictionary

static unescape_data(data)
  Un-escapes the provided bytearray data.

    Parameters  data (Bytearray) – the bytearray to unescape.

    Returns  data unescaped.

    Return type  Bytearray

static create_packet(raw, operating_mode)
  Override method.
Returns `ExplicitRXIndicatorPacket`.

Raises

- `InvalidPacketException` – if the bytearray length is less than 22. (start delim. + length (2 bytes) + frame type + 64bit addr. + 16bit addr. + source endpoint + dest. endpoint + cluster ID (2 bytes) + profile ID (2 bytes) + receive options + checksum = 22 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different than `ApiFrameType.EXPLICIT_RX_INDICATOR`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

`XBeePacket.create_packet()`
`XBeeAPIPacket._check_api_packet()`

`needs_id()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`is_broadcast()`
Override method.

See also:

`XBeeAPIPacket.is_broadcast()`

`x64bit_source_addr`  
`XBee64BitAddress` 64-bit source address.

`x16bit_source_addr`  
`XBee16BitAddress` 16-bit source address.

`receive_options`  
Integer. Receive options bitfield.

`source_endpoint`  
Integer. Source endpoint of the transmission.
**dest_endpoint**
Integer. Destination endpoint of the transmission.

**cluster_id**
Integer. Cluster ID of the transmission.

**profile_id**
Integer. Profile ID of the transmission.

**rf_data**
Bytestring. Received RF data.

digi.xbee.packets.devicecloud module

class digi.xbee.packets.devicecloud.DeviceRequestPacket (request_id, target=None, request_data=None)
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a device request packet. Packet is built using the parameters of the constructor or providing a valid API payload.

This frame type is sent out the serial port when the XBee module receives a valid device request from Device Cloud.

See also:

DeviceResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new DeviceRequestPacket object with the provided parameters.

Parameters

- **request_id** (Integer) – number that identifies the device request. (0 has no special meaning)
- **target** (String) – device request target.
- **request_data** (Bytestring, optional) – data of the request. Optional.

Raises

- **ValueError** – if request_id is less than 0 or greater than 255.
- **ValueError** – if length of target is greater than 255.

See also:

XBeeAPIPacket

static create_packet (raw, operating_mode)
Override method.

Returns DeviceRequestPacket

Raises
• \texttt{InvalidPacketException} – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + request id + transport + flags + target length + checksum = 9 bytes).

• \texttt{InvalidPacketException} – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• \texttt{InvalidPacketException} – if the first byte of ‘raw’ is not the header byte. See \texttt{SpecialByte}.

• \texttt{InvalidPacketException} – if the calculated checksum is different than the checksum field value (last byte).

• \texttt{InvalidPacketException} – if the frame type is different than \texttt{ApiFrameType.DEVICE_REQUEST}.

• \texttt{InvalidOperatingModeException} – if \texttt{operating_mode} is not supported.

See also:

\begin{verbatim}
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
\end{verbatim}

\texttt{needs_id()}

Override method.

See also:

\begin{verbatim}
XBeeAPIPacket.needs_id()
\end{verbatim}

\texttt{request_id}

Integer. Request ID of the packet.

\texttt{transport}

Integer. Transport (reserved).

\texttt{flags}

Integer. Flags (reserved).

\texttt{target}

String. Request target of the packet.

\texttt{request_data}

Bytearray. Data of the device request.

\texttt{frame_id}

Returns the frame ID of the packet.

\begin{verbatim}
Returns the frame ID of the packet.
\end{verbatim}

\texttt{Return type} Integer

\texttt{get_checksum()}

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

\begin{verbatim}
Returns checksum value of this XBeePacket.
\end{verbatim}
Return type  Integer
See also:

`factory`

get_frame_spec_data()
Override method.
See also:

`XBeePacket.get_frame_spec_data()`

get_frame_type()
Returns the frame type of this packet.

Returns  the frame type of this packet.
Return type  `ApiFrameType`
See also:

`ApiFrameType`

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns  the frame type integer value of this packet.
Return type  Integer
See also:

`ApiFrameType`

is_broadcast()
Returns whether this packet is broadcast or not.

Returns  True if this packet is broadcast, False otherwise.
Return type  Boolean

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters  escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns  raw bytearray of the XBeePacket.
Return type  Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.
Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.devicecloud.DeviceResponsePacket

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a device response packet. Packet is built using the parameters of the constructor or providing a valid API payload.

This frame type is sent to the serial port by the host in response to the DeviceRequestPacket. It should be sent within five seconds to avoid a timeout error.

See also:

DeviceRequestPacket
XBeeAPIPacket

Class constructor. Instantiates a new DeviceResponsePacket object with the provided parameters.

Parameters

• frame_id (Integer) – the frame ID of the packet.

• request_id (Integer) – device Request ID. This number should match the device request ID in the device request. Otherwise, an error will occur. (0 has no special meaning)

• response_data (Bytearray, optional) – data of the response. Optional.

Raises

• ValueError – if frame_id is less than 0 or greater than 255.

• ValueError – if request_id is less than 0 or greater than 255.

See also:

XBeeAPIPacket

static create_packet(raw, operating_mode)
Override method.

Returns DeviceResponsePacket

Raises

• InvalidPacketException – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + request id + reserved + checksum = 8 bytes).
• **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.

• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).

• **InvalidPacketException** – if the frame type is different than `ApiFrameType.DEVICE_RESPONSE`.

• **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

**needs_id()**

Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

**request_id**

Integer. Request ID of the packet.

**request_data**

Bytearray. Data of the device response.

**frame_id**

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

```python
factory
```

**get_frame_spec_data()**

Override method.

See also:
XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped(Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type(Bytearray)

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type(Dictionary)

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data(Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type(Bytearray)
class digi.xbee.packets.devicecloud.DeviceResponseStatusPacket(frame_id, status)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a device response status packet. Packet is built using the parameters of the constructor or providing a valid API payload.

This frame type is sent to the serial port after the serial port sends a DeviceResponsePacket.

See also:

DeviceResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new DeviceResponseStatusPacket object with the provided parameters.

Parameters

- **frame_id** (Integer) – the frame ID of the packet.
- **status** (DeviceCloudStatus) – device response status.

Raises **ValueError** – if frame_id is less than 0 or greater than 255.

See also:

DeviceCloudStatus
XBeeAPIPacket

static create_packet(raw, operating_mode)

Override method.

Returns DeviceResponseStatusPacket

Raises

- **InvalidPacketException** – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + device response status + checksum = 7 bytes).
- **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is different than ApiFrameType.DEVICE_RESPONSE_STATUS.
- **InvalidOperatingModeException** – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

status
DeviceCloudStatus. Status of the device response.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.
The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType
**get_frame_type_value()**

Returns the frame type integer value of this packet.

- **Returns** the frame type integer value of this packet.
- **Return type** Integer

See also:

*ApiFrameType*

**is_broadcast()**

Returns whether this packet is broadcast or not.

- **Returns** True if this packet is broadcast, False otherwise.
- **Return type** Boolean

**output** *(escaped=False)*

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

- **Parameters** escaped *(Boolean)* – indicates if the raw bytearray will be escaped or not.
- **Returns** raw bytearray of the XBeePacket.
- **Return type** Bytearray

**to_dict()**

Returns a dictionary with all information of the XBeePacket fields.

- **Returns** dictionary with all information of the XBeePacket fields.
- **Return type** Dictionary

**static unescape_data** *(data)*

Un-escapes the provided bytearray data.

- **Parameters** data *(Bytearray)* – the bytearray to unescape.
- **Returns** data unescaped.
- **Return type** Bytearray

**class** digi.xbee.packets.devicecloud.FrameErrorPacket *(frame_error)*

**Bases:** digi.xbee.packets.base.XBeeAPIPacket

This class represents a frame error packet. Packet is built using the parameters of the constructor or providing a valid API payload.

This frame type is sent to the serial port for any type of frame error.

See also:

*FrameError*
*XBeeAPIPacket*

Class constructor. Instantiates a new FrameErrorPacket object with the provided parameters.

- **Parameters** frame_error *(FrameError)* – the frame error.

See also:
FrameError
XBeeAPIPacket

static create_packet (raw, operating_mode)
Override method.

Returns FrameErrorPacket

Raises

- InvalidPacketException – if the bytearray length is less than 6. (start delim. + length (2 bytes) + frame type + frame error + checksum = 6 bytes).
- InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).
- InvalidPacketException – if the frame type is different than ApiFrameType.FRAME_ERROR.
- InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

error
FrameError. Frame error of the packet.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer
See also:

`factory`

`get_frame_spec_data()`
Override method.
See also:

`XBeePacket.get_frame_spec_data()`

`get_frame_type()`
Returns the frame type of this packet.

- **Returns** the frame type of this packet.
- **Return type** `ApiFrameType`
See also:

`ApiFrameType`

`get_frame_type_value()`
Returns the frame type integer value of this packet.

- **Returns** the frame type integer value of this packet.
- **Return type** `Integer`
See also:

`ApiFrameType`

`is_broadcast()`
Returns whether this packet is broadcast or not.

- **Returns** `True` if this packet is broadcast, `False` otherwise.
- **Return type** `Boolean`

`output` *(escaped=False)*
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

- **Parameters** `escaped` *(Boolean)* – indicates if the raw bytearray will be escaped or not.
- **Returns** raw bytearray of the XBeePacket.
- **Return type** `Bytearray`

`to_dict()`
Returns a dictionary with all information of the XBeePacket fields.

- **Returns** dictionary with all information of the XBeePacket fields.
Return type: Dictionary

**static unescape_data** *(data)*

Un-escapes the provided bytearray data.

Parameters **data** *(Bytearray)* – the bytearray to unescape.

Returns data unescaped.

Return type: Bytearray

```python
class digi.xbee.packets.devicecloud.SendDataRequestPacket(frame_id, path, content_type, options, file_data=None)
```

Bases: `digi.xbee.packets.base.XBeeAPIPacket`

This class represents a send data request packet. Packet is built using the parameters of the constructor or providing a valid API payload.

This frame type is used to send a file of the given name and type to Device Cloud.

If the frame ID is non-zero, a `SendDataResponsePacket` will be received.

See also:

`SendDataResponsePacket`

`XBeeAPIPacket`

Class constructor. Instantiates a new `SendDataRequestPacket` object with the provided parameters.

Parameters

- **frame_id** *(Integer)* – the frame ID of the packet.
- **path** *(String)* – path of the file to upload to Device Cloud.
- **content_type** *(String)* – content type of the file to upload.
- **options** *(SendDataRequestOptions)* – the action when uploading a file.
- **file_data** *(Bytearray, optional)* – data of the file to upload. Optional.

Raises `ValueError` – if frame_id is less than 0 or greater than 255.

See also:

`XBeeAPIPacket`

```python
static create_packet(raw, operating_mode)
```

Override method.

Returns `SendDataRequestPacket`

Raises

- `InvalidPacketException` – if the bytearray length is less than 10. (start delim. + length (2 bytes) + frame type + frame id + path length + content type length + transport + options + checksum = 10 bytes).
• `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.

• `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).

• `InvalidPacketException` – if the frame type is different than `ApiFrameType.SEND_DATA_REQUEST`.

• `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

```python
needs_id()
```

Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

```python
path
```

String. Path of the file to upload to Device Cloud.

```python
content_type
```

String. The content type of the file to upload.

```python
options
```

`SendDataRequestOptions`. File upload operation options.

```python
file_data
```

Bytearray. Data of the file to upload.

```python
frame_id
```

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type `Integer`

```python
get_checksum()
```

Returns the checksum value of this `XBeePacket`.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this `XBeePacket`.

Return type `Integer`

See also:

```python
factory
```
**get_frame_spec_data()**
Override method.

See also:

```
XBeePacket.get_frame_spec_data()
```

**get_frame_type()**
Returns the frame type of this packet.

- **Returns** the frame type of this packet.
- **Return type** `ApiFrameType`

See also:

```
ApiFrameType
```

**get_frame_type_value()**
Returns the frame type integer value of this packet.

- **Returns** the frame type integer value of this packet.
- **Return type** Integer

See also:

```
ApiFrameType
```

**is_broadcast()**
Returns whether this packet is broadcast or not.

- **Returns** `True` if this packet is broadcast, `False` otherwise.
- **Return type** Boolean

**output(escaped=False)**
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

- **Parameters** `escaped (Boolean)` – indicates if the raw bytearray will be escaped or not.
- **Returns** raw bytearray of the XBeePacket.
- **Return type** Bytearray

**to_dict()**
Returns a dictionary with all information of the XBeePacket fields.

- **Returns** dictionary with all information of the XBeePacket fields.
- **Return type** Dictionary

**static unescape_data(data)**
Un-escapes the provided bytearray data.

- **Parameters** `data (Bytearray)` – the bytearray to unescape.
Returns data unescaped.

Returns the frame ID of the packet.

Return type Bytearray

class digi.xbee.packets.devicecloud.SendDataResponsePacket (frame_id, status)
    Bases: digi.xbee.packets.base.XBeeAPIPacket

    This class represents a send data response packet. Packet is built using the parameters of the constructor or providing a valid API payload.

    This frame type is sent out the serial port in response to the SendDataRequestPacket, providing its frame ID is non-zero.

    See also:

    SendDataRequestPacket
    XBeeAPIPacket

Class constructor. Instantiates a new SendDataResponsePacket object with the provided parameters.

Parameters

• frame_id (Integer) – the frame ID of the packet.
• status (DeviceCloudStatus) – the file upload status.

    Raises ValueError – if frame_id is less than 0 or greater than 255.

    See also:

    DeviceCloudStatus
    XBeeAPIPacket

frame_id
    Returns the frame ID of the packet.

    Returns the frame ID of the packet.

    Return type Integer

get_checksum()
    Returns the checksum value of this XBeePacket.

    The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

    Returns checksum value of this XBeePacket.

    Return type Integer

    See also:

    factory

get_frame_spec_data()
    Override method.

    See also:
**get_frame_type()**

Returns the frame type of this packet.

- **Returns**: the frame type of this packet.
- **Return type**: `ApiFrameType`

See also:

`ApiFrameType`

**get_frame_type_value()**

Returns the frame type integer value of this packet.

- **Returns**: the frame type integer value of this packet.
- **Return type**: `Integer`

See also:

`ApiFrameType`

**is_broadcast()**

Returns whether this packet is broadcast or not.

- **Returns**: `True` if this packet is broadcast, `False` otherwise.
- **Return type**: `Boolean`

**output(escaped=False)**

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

- **Parameters**: `escaped (Boolean)` – indicates if the raw bytearray will be escaped or not.
- **Returns**: raw bytearray of the XBeePacket.
- **Return type**: `Bytearray`

**to_dict()**

Returns a dictionary with all information of the XBeePacket fields.

- **Returns**: dictionary with all information of the XBeePacket fields.
- **Return type**: `Dictionary`

**static unescape_data(data)**

Un-escapes the provided bytearray data.

- **Parameters**: `data (Bytearray)` – the bytearray to unescape.
- **Returns**: `data` unescaped.
- **Return type**: `Bytearray`

**static create_packet(raw, operating_mode)**

Override method.
Returns `SendDataResponsePacket`

 Raises

- `InvalidPacketException` – if the bytearray length is less than 10. (start delim. + length (2 bytes) + frame type + frame id + status + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different than `ApiFrameType.SEND_DATA_RESPONSE`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

`XBeePacket.create_packet()`

`XBeeAPIPacket._check_api_packet()`

`needs_id()`

Override method.

See also:

`XBeeAPIPacket.needs_id()`

`status`

`DeviceCloudStatus`. The file upload status.

digi.xbee.packets.digimesh module

class digi.xbee.packets.digimesh.RouteInformationPacket

    (src_event, timestamp, ack_timeout_count, tx_block_count, dst_addr, src_addr, responder_addr, successor_addr, additional_data=None)

Bases: `digi.xbee.packets.base.XBeeAPIPacket`

This class represents a DigiMesh Route Information packet. Packet is built using the parameters of the constructor or providing a valid API payload.

A Route Information Packet can be output for DigiMesh unicast transmissions on which the NACK enable or the Trace Route enable TX option is enabled.

See also:
Class constructor. Instantiates a new `RouteInformationPacket` object with the provided parameters.

**Parameters**

- `src_event` (*Integer*) – Source event identifier. 0x11=NACK, 0x12=Trace route
- `timestamp` (*Integer*) – System timer value on the node generating the this packet. The timestamp is in microseconds.
- `ack_timeout_count` (*Integer*) – The number of MAC ACK timeouts.
- `tx_block_count` (*Integer*) – The number of times the transmission was blocked due to reception in progress.
- `dst_addr` (*XBee64BitAddress*) – The 64-bit address of the final destination node of this network-level transmission.
- `src_addr` (*XBee64BitAddress*) – The 64-bit address of the source node of this network-level transmission.
- `responder_addr` (*XBee64BitAddress*) – The 64-bit address of the node that generates this packet after it sends (or attempts to send) the packet to the next hop (successor node).
- `successor_addr` (*XBee64BitAddress*) – The 64-bit address of the next node after the responder in the route towards the destination, whether or not the packet arrived successfully at the successor node.
- `additional_data` (*Bytearray, optional, default=`None`) – Additional data of the packet.

**Raises**

- `ValueError` – if `src_event` is not 0x11 or 0x12.
- `ValueError` – if `timestamp` is not between 0 and 0xFFFFFFFF.
- `ValueError` – if `ack_timeout_count` or `tx_block_count` are not between 0 and 255.

**See also:**

*XBee64BitAddress*

*XBeeAPIPacket*

**static create_packet**(raw, operating_mode)

Override method.

**Returns** `RouteInformationPacket`.

**Raises**

- `InvalidPacketException` – If the bytearray length is less than 46. (start delim. + length (2 bytes) + frame type + src_event + length + timestamp (4 bytes) + ack timeout count + tx block count + reserved + dest addr (8 bytes) + src addr (8 bytes) + responder addr (8 bytes) + successor addr (8 bytes) + checksum = 46 bytes).
• **InvalidPacketException** – If the length field of `raw` is different from its real length. (length field: bytes 1 and 3)

• **InvalidPacketException** – If the first byte of ‘raw’ is not the header byte. See `SpecialByte`.

• **InvalidPacketException** – If the calculated checksum is different from the checksum field value (last byte).

• **InvalidPacketException** – If the frame type is not `ApiFrameType.DIGIMESH_ROUTE_INFORMATION`.

• **InvalidPacketException** – If the internal length byte of the rest of the frame (without the checksum) is different from its real length.

• **InvalidOperatingModeException** – If `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

**needs_id()**

Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

**src_event**

Returns the source event.

Returns: The source event.

Return type: Integer

**length**

Returns the number of bytes that follow, excluding the checksum.

Returns: Data length.

Return type: Integer

**timestamp**

Returns the system timer value on the node generating this package. The timestamp is in microseconds.

Returns: The system timer value in microseconds.

Return type: Integer

**ack_timeout_count**

Returns the number of MAC ACK timeouts that occur.

Returns: The number of MAC ACK timeouts that occur.

Return type: Integer

**tx_block_count**

Returns the number of times the transmission was blocked due to reception in progress.
Returns

The number of times the transmission was blocked due to reception in progress.

Return type  Integer

frame_id
Returns the frame ID of the packet.

Returns  the frame ID of the packet.

Return type  Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns  checksum value of this XBeePacket.

Return type  Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns  the frame type of this packet.

Return type  ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns  the frame type integer value of this packet.

Return type  Integer

See also:

ApiFrameType
is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.
Return type Boolean

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.
Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.
Return type Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.
Return type Bytearray

dst_addr
Returns the 64-bit source address.

Returns
The 64-bit address of the final destination node.
Return type XBee64BitAddress

See also:

XBee64BitAddress

src_addr
Returns the 64-bit address of the source node of this network-level transmission.

Returns The 64-bit address of the source node.
Return type XBee64BitAddress

See also:

XBee64BitAddress

responder_addr
Returns the 64-bit address of the node that generates this packet after it sends (or attempts to send) the packet to the next hop (successor node).

Returns The 64-bit address of the responder node.
Return type  XBee64BitAddress
See also:

XBee64BitAddress

successor_addr
Returns the 64-bit address of the next node after the responder in the route towards the destination, whether or not the packet arrived successfully at the successor node.

Returns  The 64-bit address of the successor node.

Return type  XBee64BitAddress
See also:

XBee64BitAddress

digi.xbee.packets.network module
class  digi.xbee.packets.network.RXIPv4Packet(source_address, dest_port, source_port, ip_protocol, data=None)
Bases: digi.xbee.packets.base.XBeeAPIPacket
This class represents an RX (Receive) IPv4 packet. Packet is built using the parameters of the constructor or providing a valid byte array.

See also:

TXIPv4Packet
XBeeAPIPacket

Class constructor. Instantiates a new RXIPv4Packet object with the provided parameters.

Parameters

• source_address (IPv4Address) – IPv4 address of the source device.
• dest_port (Integer) – destination port number.
• source_port (Integer) – source port number.
• ip_protocol (IPProtocol) – IP protocol used for transmitted data.
• data (Bytearray, optional) – data that is sent to the destination device. Optional.

Raises

• ValueError – if dest_port is less than 0 or greater than 65535 or
• ValueError – if source_port is less than 0 or greater than 65535.

See also:

IPProtocol
static create_packet (raw, operating_mode)
  Override method.

  Returns  RXIPv4Packet.

  Raises

  • InvalidPacketException – if the bytearray length is less than 15. (start delim +
    length (2 bytes) + frame type + source address (4 bytes) + dest port (2 bytes) + source port
    (2 bytes) + network protocol + status + checksum = 15 bytes)

  • InvalidPacketException – if the length field of raw is different than its real
    length. (length field: bytes 2 and 3)

  • InvalidPacketException – if the first byte of raw is not the header byte. See
    SPECIAL_BYTE.

  • InvalidPacketException – if the calculated checksum is different than the check-
    sum field value (last byte).

  • InvalidPacketException – if the frame type is not ApiFrameType.RX_IPV4.

  • InvalidOperatingModeException – if operating_mode is not supported.

  See also:

  XBeePacket.create_packet()
  XBeeAPIPacket._check_api_packet()

needs_id()
  Override method.

  See also:

  XBeeAPIPacket.needs_id()

source_address
  ipaddress.IPv4Address. IPv4 address of the source device.

dest_port
  Integer. Destination port.

source_port
  Integer. Source port.

ip_protocol
  IPProtocol. IP protocol used in the transmission.

data
  Bytarray. Data of the packet.

frame_id
  Returns the frame ID of the packet.

  Returns  the frame ID of the packet.

  Return type  Integer
**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

**Returns** checksum value of this XBeePacket.

**Return type** Integer

**See also:**

*factory*

**get_frame_spec_data()**

Override method.

**See also:**

*XBeePacket.get_frame_spec_data()*

**get_frame_type()**

Returns the frame type of this packet.

**Returns** the frame type of this packet.

**Return type** ApiFrameType

**See also:**

*ApiFrameType*

**get_frame_type_value()**

Returns the frame type integer value of this packet.

**Returns** the frame type integer value of this packet.

**Return type** Integer

**See also:**

*ApiFrameType*

**is_broadcast()**

Returns whether this packet is broadcast or not.

**Returns** True if this packet is broadcast, False otherwise.

**Return type** Boolean

**output**(escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

**Parameters** escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.network.TXIPv4Packet(frame_id, dest_address, dest_port, source_port, ip_protocol, transmit_options, data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an TX (Transmit) IPv4 packet. Packet is built using the parameters of the constructor or providing a valid byte array.

See also:

RXIPv4Packet
XBeeAPIPacket

Class constructor. Instantiates a new TXIPv4Packet object with the provided parameters.

Parameters

• frame_id (Integer) – the frame ID. Must be between 0 and 255.
• dest_address (IPv4Address) – IPv4 address of the destination device.
• dest_port (Integer) – destination port number.
• source_port (Integer) – source port number.
• ip_protocol (IPProtocol) – IP protocol used for transmitted data.
• transmit_options (Integer) – the transmit options of the packet.
• data (Bytearray, optional) – data that is sent to the destination device. Optional.

Raises

• ValueError – if frame_id is less than 0 or greater than 255.
• ValueError – if dest_port is less than 0 or greater than 65535.
• ValueError – if source_port is less than 0 or greater than 65535.

See also:

IPProtocol
OPTIONS_CLOSE_SOCKET = 2
This option will close the socket after the transmission.

OPTIONS_LEAVE_SOCKET_OPEN = 0
This option will leave socket open after the transmission.

static create_packet (raw, operating_mode)
Override method.

Returns TXIPv4Packet.

Raises

• InvalidPacketException – if the bytearray length is less than 16. (start delim + length (2 bytes) + frame type + frame id + dest address (4 bytes) + dest port (2 bytes) + source port (2 bytes) + network protocol + transmit options + checksum = 16 bytes)

• InvalidPacketException – if the length field of raw is different than its real length. (length field: bytes 2 and 3)

• InvalidPacketException – if the first byte of raw is not the header byte. See SPECIAL_BYTE.

• InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

• InvalidPacketException – if the frame type is not ApiFrameType.TX_IPV4.

• InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet ()
XBeeAPIPacket._check_api_packet ()

needs_id ()
Override method.

See also:

XBeeAPIPacket.needs_id ()

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum ()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:
factory

get_frame_spec_data()
    Override method.
    See also:

    XBeePacket.get_frame_spec_data()

get_frame_type()
    Returns the frame type of this packet.
    Returns the frame type of this packet.
    Return type ApiFrameType
    See also:

    ApiFrameType

get_frame_type_value()
    Returns the frame type integer value of this packet.
    Returns the frame type integer value of this packet.
    Return type Integer
    See also:

    ApiFrameType

is_broadcast()
    Returns whether this packet is broadcast or not.
    Returns True if this packet is broadcast, False otherwise.
    Return type Boolean

output(escaped=False)
    Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.
    Parameters escaped(Boolean) – indicates if the raw bytearray will be escaped or not.
    Returns raw bytearray of the XBeePacket.
    Return type bytearray

to_dict()
    Returns a dictionary with all information of the XBeePacket fields.
    Returns dictionary with all information of the XBeePacket fields.
    Return type Dictionary
**static unescape_data**(data)

Un-escapes the provided bytearray data.

**Parameters**

- **data**(Bytearray) – the bytearray to unescape.

**Returns**

data unescaped.

**Return type**

Bytearray

**dest_address**

ipaddress.IPv4Address. IPv4 address of the destination device.

**dest_port**

Integer. Destination port.

**source_port**

Integer. Source port.

**ip_protocol**

IPProtocol. IP protocol.

**transmit_options**

Integer. Transmit options.

**data**

Bytearray. Data of the packet.

---

**digi.xbee.packets.raw module**

**class**

digi.xbee.packets.raw.TX64Packet(frame_id, x64bit_addr, transmit_options, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a TX (Transmit) 64 Request packet. Packet is built using the parameters of the constructor or providing a valid byte array.

A TX Request message will cause the module to transmit data as an RF Packet.

**See also:**

XBeeAPIPacket

Class constructor. Instantiates a new TX64Packet object with the provided parameters.

**Parameters**

- **frame_id**(Integer) – the frame ID of the packet.
- **x64bit_addr**(XBee64BitAddress) – the 64-bit destination address.
- **transmit_options**(Integer) – bitfield of supported transmission options.
- **rf_data**(Bytearray, optional) – RF data that is sent to the destination device. Optional.

**See also:**

TransmitOptions
XBee64BitAddress
XBeeAPIPacket

Raises ValueError – if frame_id is less than 0 or greater than 255.

static create_packet (raw, operating_mode)
Override method.

Returns TX64Packet.

Raises

• InvalidPacketException – if the bytearray length is less than 15. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + transmit options + checksum = 15 bytes).

• InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.

• InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

• InvalidPacketException – if the frame type is different than ApiFrameType.TX_64.

• InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet ()
XBeeAPIPacket._check_api_packet ()

needs_id ()
Override method.

See also:

XBeeAPIPacket.needs_id ()

x64bit_dest_addr
XBee64BitAddress. 64-bit destination address.

transmit_options
Integer. Transmit options bitfield.

rf_data
Bytearray. RF data to send.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer
get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.

Return type: Bytearray

to_dict()

Returns a dictionary with all information of the XBeePacket fields.

Returns: dictionary with all information of the XBeePacket fields.

Return type: Dictionary

static unescape_data(data)

Un-escapes the provided bytearray data.

Parameters: data (Bytearray) – the bytearray to unescape.

Returns: data unescaped.

Return type: Bytearray

class digi.xbee.packets.raw.TX16Packet(frame_id, x16bit_addr, transmit_options, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a TX (Transmit) 16 Request packet. Packet is built using the parameters of the constructor or providing a valid byte array.

A TX request message will cause the module to transmit data as an RF packet.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new TX16Packet object with the provided parameters.

Parameters:

- **frame_id** (Integer) – the frame ID of the packet.
- **x16bit_addr** (XBee16BitAddress) – the 16-bit destination address.
- **transmit_options** (Integer) – bitfield of supported transmission options.
- **rf_data** (Bytearray, optional) – RF data that is sent to the destination device. Optional.

See also:

TransmitOptions
XBee16BitAddress
XBeeAPIPacket

Raises: ValueError – if frame_id is less than 0 or greater than 255.

static create_packet(raw, operating_mode)

Override method.

Returns: TX16Packet.

Raises: None.
InvalidPacketException – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + frame id + 16bit addr. + transmit options + checksum = 9 bytes).

InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.

InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

InvalidPacketException – if the frame type is different than ApiFrameType. TX_16.

InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

x16bit_dest_addr
XBee64BitAddress. 16-bit destination address.

transmit_options
Integer. Transmit options bitfield.

rf_data
Bytarray. RF data to send.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:
factory

get_frame_spec_data()
   Override method.

   See also:

   XBeePacket.get_frame_spec_data()

get_frame_type()
   Returns the frame type of this packet.

   Returns  the frame type of this packet.
   Return type  ApiFrameType

   See also:

   ApiFrameType

get_frame_type_value()
   Returns the frame type integer value of this packet.

   Returns  the frame type integer value of this packet.
   Return type  Integer

   See also:

   ApiFrameType

is_broadcast()
   Returns whether this packet is broadcast or not.

   Returns  True if this packet is broadcast, False otherwise.
   Return type  Boolean

output (escaped=False)
   Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

   Parameters  escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

   Returns  raw bytearray of the XBeePacket.
   Return type  Bytearray

to_dict()
   Returns a dictionary with all information of the XBeePacket fields.

   Returns  dictionary with all information of the XBeePacket fields.
   Return type  Dictionary
static unescape_data\(data\)
Un-escapes the provided bytearray data.

Parameters \(data\) (Bytearray) – the bytearray to unescape.

Returns \(data\) unescaped.

Return type Bytearray

class digi.xbee.packets.raw.TXStatusPacket \((frame_id, transmit_status)\)
Bases: digi.xbee.packets.base.XBeeAPIPacket
This class represents a TX (Transmit) status packet. Packet is built using the parameters of the constructor or providing a valid API payload.
When a TX request is completed, the module sends a TX status message. This message will indicate if the packet was transmitted successfully or if there was a failure.

See also:

TX16Packet
TX64Packet
XBeeAPIPacket

Class constructor. Instantiates a new TXStatusPacket object with the provided parameters.

Parameters

- \(frame_id\) (Integer) – the frame ID of the packet.
- \(transmit_status\) (TransmitStatus) – transmit status. Default: SUCCESS.

Raises ValueError – if \(frame_id\) is less than 0 or greater than 255.

See also:

TransmitStatus
XBeeAPIPacket

static create_packet\(raw, operating_mode\)
Override method.

Returns TXStatusPacket.

Raises

- InvalidPacketException – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + transmit status + checksum = 7 bytes).
- InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).
• \textit{InvalidPacketException} – if the frame type is different than \texttt{ApiFrameType.TX\_16}.
• \textit{InvalidOperatingModeException} – if \texttt{operating\_mode} is not supported.

See also:

\begin{verbatim}
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
\end{verbatim}

\texttt{needs\_id()}
Override method.

See also:

\begin{verbatim}
XBeeAPIPacket.needs\_id()
\end{verbatim}

\texttt{transmit\_status}
\texttt{TransmitStatus}. Transmit status.

\texttt{frame\_id}
Returns the frame ID of the packet.

\begin{verbatim}
Returns the frame ID of the packet.
Return type Integer
\end{verbatim}

\texttt{get\_checksum()}
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

\begin{verbatim}
Returns checksum value of this XBeePacket.
Return type Integer
\end{verbatim}

See also:

\begin{verbatim}
factory
\end{verbatim}

\texttt{get\_frame\_spec\_data()}
Override method.

See also:

\begin{verbatim}
XBeePacket.get\_frame\_spec\_data()
\end{verbatim}

\texttt{get\_frame\_type()}
Returns the frame type of this packet.

\begin{verbatim}
Returns the frame type of this packet.
Return type \texttt{ApiFrameType}
\end{verbatim}
See also:

(ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

(ApiFrameType

is.broadcast()

Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type bytearray

to_dict()

Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)

Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type bytearray

class digi.xbee.packets.raw.RX64Packet(x64bit_addr, rssi, receive_options, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an RX (Receive) 64 request packet. Packet is built using the parameters of the constructor or providing a valid API byte array.

When the module receives an RF packet, it is sent out the UART using this message type.

This packet is the response to TX (transmit) 64 request packets.

See also:
ReceiveOptions
TX64Packet
XBeeAPIPacket

Class constructor. Instantiates a `RX64Packet` object with the provided parameters.

**Parameters**

- `x64bit_addr` (`XBee64BitAddress`) – the 64-bit source address.
- `rssi` (`Integer`) – received signal strength indicator.
- `receive_options` (`Integer`) – bitfield indicating the receive options.
- `rf_data` (`Bytearray, optional`) – received RF data. Optional.

See also:

ReceiveOptions
XBee64BitAddress
XBeeAPIPacket

**static create_packet** (`raw, operating_mode`)

Override method.

Returns `RX64Packet`

Raises

- `InvalidPacketException` – if the bytearray length is less than 15. (start delim. + length (2 bytes) + frame type + 64bit addr. + rssi + receive options + checksum = 15 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different than `ApiFrameType.RX_64`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

`XBeePacket.create_packet()`
`XBeeAPIPacket._check_api_packet()`

**needs_id**

Override method.

See also:
XBeeAPIPacket.needs_id()

is_broadcast()
    Override method.
    See also:

XBeeAPIPacket.is_broadcast()

x64bit_source_addr
    XBee64BitAddress. 64-bit source address.

rssi
    Integer. Received Signal Strength Indicator (RSSI) value.

receive_options
    Integer. Receive options bitfield.

rf_data
    Bytestring. Received RF data.

frame_id
    Returns the frame ID of the packet.

    Returns the frame ID of the packet.
    Return type Integer

get_checksum()
    Returns the checksum value of this XBeePacket.

    The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

    Returns checksum value of this XBeePacket.
    Return type Integer
    See also:

factory

get_frame_spec_data()
    Override method.
    See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
    Returns the frame type of this packet.

    Returns the frame type of this packet.
    Return type ApiFrameType
See also:

 ApiServiceType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

output(escaped=False)

Returns the raw bytarray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytarray will be escaped or not.

Returns raw bytarray of the XBeePacket.

Return type Bytearray

to_dict()

Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)

Un-escapes the provided bytarray data.

Parameters data (Bytearray) – the bytarray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.raw.RX16Packet(x16bit_addr, rssi, receive_options, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an RX (Receive) 16 Request packet. Packet is built using the parameters of the constructor or providing a valid API byte array.

When the module receives an RF packet, it is sent out the UART using this message type

This packet is the response to TX (Transmit) 16 Request packets.

See also:

ReceiveOptions
TX16Packet
XBeeAPIPacket

Class constructor. Instantiates a RX16Packet object with the provided parameters.
Parameters

- `x16bit_addr (XBee16BitAddress)` – the 16-bit source address.
- `rssi (Integer)` – received signal strength indicator.
- `receive_options (Integer)` – bitfield indicating the receive options.
- `rf_data (Bytearray, optional)` – received RF data. Optional.

See also:

- `ReceiveOptions`
- `XBee16BitAddress`
- `XBeeAPIPacket`

**static create_packet (raw, operating_mode)**

Override method.

Returns `RX16Packet`.

Raises

- `InvalidPacketException` – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + 16bit addr. + rssi + receive options + checksum = 9 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different than `ApiFrameType.RX_16`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

- `XBeePacket.create_packet()`
- `XBeeAPIPacket._check_api_packet()`

`needs_id()`

Override method.

See also:

- `XBeeAPIPacket.needs_id()`

`is_broadcast()`

Override method.
See also:
XBeeAPIMessage.is_broadcast()

\texttt{x16bit_{source\_addr}}
\begin{itemize}
\item \texttt{XBee16BitAddress}. 16-bit source address.
\end{itemize}

\texttt{rssi}
\begin{itemize}
\item integer. Received Signal Strength Indicator (RSSI) value.
\end{itemize}

\texttt{receive\_options}
\begin{itemize}
\item integer. Receive options bitfield.
\end{itemize}

\texttt{rf\_data}
\begin{itemize}
\item bytearray. Received RF data.
\end{itemize}

\texttt{frame\_id}
\begin{itemize}
\item Returns the frame ID of the packet.
\item \texttt{Returns} the frame ID of the packet.
\item \texttt{Return type} integer
\end{itemize}

\texttt{get\_checksum()} 
\begin{itemize}
\item Returns the checksum value of this XBeePacket.
\item The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
\item \texttt{Returns} checksum value of this XBeePacket.
\item \texttt{Return type} integer
\end{itemize}

See also:

\texttt{factory}

\texttt{get\_frame\_spec\_data()} 
\begin{itemize}
\item Override method.
\end{itemize}

See also:

\texttt{XBeePacket.get\_frame\_spec\_data()}

\texttt{get\_frame\_type()} 
\begin{itemize}
\item Returns the frame type of this packet.
\item \texttt{Returns} the frame type of this packet.
\item \texttt{Return type} \texttt{ApiFrameType}
\end{itemize}

See also:

\texttt{ApiFrameType}
get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.
Return type Integer
See also:

ApiFrameType

output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.
Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.
Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.
Return type Bytearray

class digi.xbee.packets.raw.RX64IOPacket (x64bit_addr, rssi, receive_options, rf_data)
Bases: digi.xbee.packets.base.XBeeAPIPacket
This class represents an RX64 address IO packet. Packet is built using the parameters of the constructor or
providing a valid API payload.
I/O data is sent out the UART using an API frame.
See also:

XBeeAPIPacket

Class constructor. Instantiates an RX64IOPacket object with the provided parameters.

Parameters

• x64bit_addr (XBee64BitAddress) – the 64-bit source address.
• rssi (Integer) – received signal strength indicator.
• receive_options (Integer) – bitfield indicating the receive options.
• rf_data (Bytearray) – received RF data.

See also:
static create_packet (raw, operating_mode)

Override method.

Returns RX64IOPacket.

Raises

- InvalidPacketException – if the bytearray length is less than 20. (start delim. + length (2 bytes) + frame type + 64bit addr. + rssi + receive options + rf data (5 bytes) + checksum = 20 bytes)

- InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

- InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.

- InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

- InvalidPacketException – if the frame type is different than ApiFrameType.RX_IO_64.

- InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

is_broadcast()

Override method.

See also:

XBeeAPIPacket.is_broadcast()

x64bit_source_addr

XBee64BitAddress. 64-bit source address.
**rssi**
Integer. Received Signal Strength Indicator (RSSI) value.

**receive_options**
Integer. Receive options bitfield.

**rf_data**
Bytearray. Received RF data.

**io_sample**
IO sample corresponding to the data contained in the packet.
Type **IOSample**

**frame_id**
Returns the frame ID of the packet.

  **Returns** the frame ID of the packet.
  **Return type** Integer

**get_checksum()**
Returns the checksum value of this XBeePacket.

  The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

  **Returns** checksum value of this XBeePacket.
  **Return type** Integer

**factory**

**get_frame_spec_data()**
Override method.

  **See also:**
  XBeePacket.get_frame_spec_data()

**get_frame_type()**
Returns the frame type of this packet.

  **Returns** the frame type of this packet.
  **Return type** ApiFrameType

  **See also:**
  ApiFrameType

**get_frame_type_value()**
Returns the frame type integer value of this packet.

  **Returns** the frame type integer value of this packet.
Return type Integer

See also:

 ApiServiceType

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
 Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict ()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.raw.RX16IOPacket (x16bit_addr, rssi, receive_options, rf_data)
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents an RX16 address IO packet. Packet is built using the parameters of the constructor or providing a valid byte array.

I/O data is sent out the UART using an API frame.

See also:

XBeeAPIPacket

Class constructor. Instantiates an RX16IOPacket object with the provided parameters.

Parameters

• x16bit_addr (XBee16BitAddress) – the 16-bit source address.

• rssi (Integer) – received signal strength indicator.

• receive_options (Integer) – bitfield indicating the receive options.

• rf_data (Bytearray) – received RF data.

See also:

ReceiveOptions
XBee16BitAddress
XBeeAPIPacket

**static create_packet** *(raw, operating_mode)*

Override method.

**Returns** RX16IOPacket.

**Raises**

- *InvalidPacketException* – if the bytearray length is less than 14. (start delim. + length (2 bytes) + frame type + 16bit addr. + rssi + receive options + rf data (5 bytes) + checksum = 14 bytes).
- *InvalidPacketException* – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- *InvalidPacketException* – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- *InvalidPacketException* – if the calculated checksum is different than the checksum field value (last byte).
- *InvalidPacketException* – if the frame type is different than `ApiFrameType.RX_10_16`.
- *InvalidOperatingModeException* – if operating_mode is not supported.

See also:

- `XBeePacket.create_packet()`
- `XBeeAPIPacket._check_api_packet()`

**frame_id**

Returns the frame ID of the packet.

**Returns** the frame ID of the packet.

**Return type** Integer

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

**Returns** checksum value of this XBeePacket.

**Return type** Integer

See also:

**factory**

**get_frame_spec_data()**

Override method.

See also:
get_frame_type()  
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()  
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)  
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

needs_id()  
Override method.

See also:

XBeeAPIPacket.needs_id()
is_broadcast()
   Override method.

See also:

XBeeAPIPacket.is_broadcast()

x16bit_source_addr
   XBee16BitAddress. 16-bit source address.

rssi
   Integer. Received Signal Strength Indicator (RSSI) value.

receive_options
   Integer. Receive options bitfield.

rf_data
   Bytarray. Received RF data.

io_sample
   IO sample corresponding to the data contained in the packet.
   
   Type IOSample

digi.xbee.packets.relay module

class digi.xbee.packets.relay.UserDataRelayPacket (frame_id, local_interface, data=None)
   Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a User Data Relay packet. Packet is built using the parameters of the constructor.

The User Data Relay packet allows for data to come in on an interface with a designation of the target interface for the data to be output on.

The destination interface must be one of the interfaces found in the corresponding enumerator (see XBeeLocalInterface).

See also:

UserDataRelayOutputPacket
XBeeAPIPacket
XBeeLocalInterface

Class constructor. Instantiates a new UserDataRelayPacket object with the provided parameters.

Parameters
   • frame_id (integer) – the frame ID of the packet.
   • local_interface (XBeeLocalInterface) – the destination interface.
   • data (Bytarray, optional) – Data to send to the destination interface.

See also:
XBeeAPIPacket
XBeeLocalInterface

Raises

- `ValueError` — if `local_interface` is `None`.
- `ValueError` — if `frame_id` is less than 0 or greater than 255.

**static create_packet** *(raw, operating_mode)*

Override method.

Returns `UserDataRelayPacket`.

Raises

- `InvalidPacketException` — if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + relay interface + checksum = 7 bytes).
- `InvalidPacketException` — if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` — if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` — if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` — if the frame type is not `ApiFrameType.USER_DATA_RELAY_REQUEST`.
- `InvalidOperatingModeException` — if `operating_mode` is not supported.

See also:

`XBeePacket.create_packet()` `XBeeAPIPacket._check_api_packet()`

**needs_id()**

Override method.

See also:

`XBeeAPIPacket.needs_id()`

dest_interface

*XBeeLocalInterface*. Destination local interface.

data

Bytearray. Data to send.

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer
get_checksum()  
Returns the checksum value of this XBeePacket.  
The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.  
Returns checksum value of this XBeePacket.  
Return type Integer  
See also:  

factory

get_frame_spec_data()  
Override method.  
See also:  

XBeePacket.get_frame_spec_data()  

get_frame_type()  
Returns the frame type of this packet.  
Returns the frame type of this packet.  
Return type ApiFrameType  
See also:  

ApiFrameType

get_frame_type_value()  
Returns the frame type integer value of this packet.  
Returns the frame type integer value of this packet.  
Return type Integer  
See also:  

ApiFrameType

is_broadcast()  
Returns whether this packet is broadcast or not.  
Returns True if this packet is broadcast, False otherwise.  
Return type Boolean

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.  
Parameters escaped(Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.

Return type Bytearray
to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.
Return type Dictionary
static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.
Returns data unescaped.
Return type Bytearray
class digi.xbee.packets.relay.UserDataRelayOutputPacket (local_interface, data=None)
Bases: digi.xbee.packets.base.XBeeAPIPacket
This class represents a User Data Relay Output packet. Packet is built using the parameters of the constructor.
The User Data Relay Output packet can be received from any relay interface.
The source interface must be one of the interfaces found in the corresponding enumerator (see XBeeLocalInterface).
See also:

UserDataRelayPacket
XBeeAPIPacket
XBeeLocalInterface

Class constructor. Instantiates a new UserDataRelayOutputPacket object with the provided parameters.

Parameters

• local_interface (XBeeLocalInterface) – the source interface.
• data (Bytearray, optional) – Data received from the source interface.

Raises ValueError – if local_interface is None.
See also:

XBeeAPIPacket
XBeeLocalInterface

static create_packet (raw, operating_mode)
Override method.

Returns UserDataRelayOutputPacket.
Raises
• `InvalidPacketException` – if the bytearray length is less than 6. (start delim. + length (2 bytes) + frame type + relay interface + checksum = 6 bytes).

• `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.

• `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).

• `InvalidPacketException` – if the frame type is not `ApiFrameType.USER_DATA_RELAY_OUTPUT`.

• `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

`needs_id()`  
Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

`frame_id`  
Returns the frame ID of the packet.

- **Returns** the frame ID of the packet.
- **Return type** `Integer`

`get_checksum()`  
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

- **Returns** checksum value of this XBeePacket.
- **Return type** `Integer`

See also:

```python
factory
```

`get_frame_spec_data()`  
Override method.

See also:
XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType
See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer
See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean
output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray
to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary
static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray
src_interface
XBeeLocalInterface. Source local interface.
**data**

Bytearray. Received data.

---

**digi.xbee.packets.socket module**

**class** `digi.xbee.packets.socket.SocketCreatePacket (frame_id, protocol)`

Bases: `digi.xbee.packets.base.XBeeAPIPacket`

This class represents a Socket Create packet. Packet is built using the parameters of the constructor.

Use this frame to create a new socket with the following protocols: TCP, UDP, or TLS.

See also:

`SocketCreateResponsePacket`

`XBeeAPIPacket`

Class constructor. Instantiates a new `SocketCreatePacket` object with the provided parameters.

**Parameters**

- `frame_id` (**Integer**) – the frame ID of the packet.
- `protocol` (**IPProtocol**) – the protocol used to create the socket.

See also:

`XBeeAPIPacket`

`IPProtocol`

Raises `ValueError` – if `frame_id` is less than 0 or greater than 255.

**static create_packet (raw, operating_mode)**

Override method.

**Returns** `SocketCreatePacket`.

Raises

- `InvalidPacketException` – if thebyterray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + protocol + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_CREATE`.
- `InvalidPacketException` – if operating_mode is not supported.

See also:
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.
See also:

XBeeAPIPacket.needs_id()

protocol
IPProtocol. Communication protocol.

frame_id
Returns the frame ID of the packet.
  Returns the frame ID of the packet.
  Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.
The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
  Returns checksum value of this XBeePacket.
  Return type Integer
See also:

factory

get_frame_spec_data()
Override method.
See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.
  Returns the frame type of this packet.
  Return type ApiFrameType
See also:

ApiFrameType
get_frame_type_value()  
Returns the frame type integer value of this packet.

    Returns  the frame type integer value of this packet.
    Return type  Integer
See also:

    ApiFrameType

is_broadcast()  
Returns whether this packet is broadcast or not.

    Returns  True if this packet is broadcast, False otherwise.
    Return type  Boolean

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

    Parameters  escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
    Returns  raw bytearray of the XBeePacket.
    Return type  Bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.

    Returns  dictionary with all information of the XBeePacket fields.
    Return type  Dictionary

static unescape_data(data)  
Un-escapes the provided bytearray data.

    Parameters  data (Bytearray) – the bytearray to unescape.
    Returns  data unescaped.
    Return type  Bytearray

class  
digi.xbee.packets.socket.SocketCreateResponsePacket (frame_id,   socket_id,   status)
Bases: digi.xbee.packets.base.XBeeAPIPacket
This class represents a Socket Create Response packet. Packet is built using the parameters of the constructor.

The device sends this frame in response to a Socket Create (0x40) frame. It contains a socket ID that should be used for future transactions with the socket and a status field.

If the status field is non-zero, which indicates an error, the socket ID will be set to 0xFF and the socket will not be opened.

See also:

    SocketCreatePacket
    XBeeAPIPacket
Class constructor. Instantiates a new `SocketCreateResponsePacket` object with the provided parameters.

**Parameters**

- `frame_id (Integer)` – the frame ID of the packet.
- `socket_id (Integer)` – the unique socket ID to address the socket.
- `status (SocketStatus)` – the socket create status.

**See also:**

- `XBeeAPIPacket`
- `SocketStatus`

**Raises**

- `ValueError` – if `frame_id` is less than 0 or greater than 255.
- `ValueError` – if `socket_id` is less than 0 or greater than 255.

**static create_packet (raw, operating_mode)**

Override method.

**Returns** `SocketCreateResponsePacket`.

**Raises**

- `InvalidPacketException` – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + status + checksum = 8 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType`.`SOCKET_CREATE_RESPONSE`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

**See also:**

- `XBeePacket.create_packet()`
- `XBeeAPIPacket._check_api_packet()`

**needs_id()**

Override method.

**See also:**

- `XBeeAPIPacket.needs_id()`
socket_id
    Integer. Socket ID.

status
    SocketStatus. Socket create status.

frame_id
    Returns the frame ID of the packet.
    Returns the frame ID of the packet.
    Return type Integer

get_checksum()
    Returns the checksum value of this XBeePacket.
    The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
    Returns checksum value of this XBeePacket.
    Return type Integer
    See also:

    factory

get_frame_spec_data()
    Override method.
    See also:

    XBeePacket.get_frame_spec_data()

get_frame_type()
    Returns the frame type of this packet.
    Returns the frame type of this packet.
    Return type ApiFrameType
    See also:

    ApiFrameType

get_frame_type_value()
    Returns the frame type integer value of this packet.
    Returns the frame type integer value of this packet.
    Return type Integer
    See also:

    ApiFrameType
is_broadcast()  
Returns whether this packet is broadcast or not.
- **Returns** True if this packet is broadcast, False otherwise.
- **Return type** Boolean

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.
- **Parameters** escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
- **Returns** raw bytearray of the XBeePacket.
- **Return type** Bytearray

**to_dict()**  
Returns a dictionary with all information of the XBeePacket fields.
- **Returns** dictionary with all information of the XBeePacket fields.
- **Return type** Dictionary

static unescape_data(data)  
Un-escapes the provided bytearray data.
- **Parameters** data (Bytearray) – the bytearray to unescape.
- **Returns** data unescaped.
- **Return type** Bytearray

```python
class digi.xbee.packets.socket.SocketOptionRequestPacket(frame_id, socket_id, option, option_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket
```

This class represents a Socket Option Request packet. Packet is built using the parameters of the constructor.
Use this frame to modify the behavior of sockets to be different from the normal default behavior.
If the Option Data field is zero-length, the Socket Option Response Packet (0xC1) reports the current effective value.

**See also:**

SocketOptionResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new SocketOptionRequestPacket object with the provided parameters.

**Parameters**
- **frame_id** (Integer) – the frame ID of the packet.
- **socket_id** (Integer) – the socket ID to modify.
- **option** (SocketOption) – the socket option of the parameter to change.
- **option_data** (Bytearray, optional) – the option data. Optional.

**See also:**

XBeeAPIPacket
SocketOption

Raises

- **ValueError** – if `frame_id` is less than 0 or greater than 255.
- **ValueError** – if `socket_id` is less than 0 or greater than 255.

```python
static create_packet (raw, operating_mode)
```

Override method.

Returns :class:SocketOptionRequestPacket.

Raises

- **InvalidPacketException** – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + option + checksum = 8 bytes).
- **InvalidPacketException** – if the length field of `raw` is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of `raw` is not the header byte. See SpecialByte.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is not ApiFrameType.SOCKEYT_OPTION_REQUEST.
- **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

```
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

```python
needs_id ()
```

Override method.

See also:

```
XBeeAPIPacket.needs_id()
```

```python
socket_id
```

Integer. Socket ID.

```python
option
```

:paramref:SocketOption. Socket option.

```python
option_data
```

Bytearray. Socket option data.

```python
frame_id
```

Returns the frame ID of the packet.

Returns the frame ID of the packet.
Return type    Integer

**get_checksum()**
Returns the checksum value of this XBeePacket.
The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns    checksum value of this XBeePacket.
Return type    Integer

See also:

```python
factory
```

**get_frame_spec_data()**
Override method.

See also:

```python
XBeePacket.get_frame_spec_data()
```

**get_frame_type()**
Returns the frame type of this packet.

Returns    the frame type of this packet.
Return type    ApiFrameType

See also:

```python
ApiFrameType
```

**get_frame_type_value()**
Returns the frame type integer value of this packet.

Returns    the frame type integer value of this packet.
Return type    Integer

See also:

```python
ApiFrameType
```

**is_broadcast()**
Returns whether this packet is broadcast or not.

Returns    True if this packet is broadcast, False otherwise.
Return type    Boolean

**output(escaped=False)**
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.
Parameters `escaped` (`Boolean`) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(`data`)
Un-escapes the provided bytearray data.

Parameters `data` (`Bytearray`) – the bytearray to unescape.

Returns `data` unescaped.

Return type Bytearray

class digi.xbee.packets.socket.SocketOptionResponsePacket (`frame_id`, `socket_id`, `option`, `status`, `option_data=None`)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Option Response packet. Packet is built using the parameters of the constructor.

Reports the status of requests made with the Socket Option Request (0x41) packet.

See also:

SocketOptionRequestPacket
XBeeAPIPacket

Class constructor. Instantiates a new `SocketOptionResponsePacket` object with the provided parameters.

Parameters

• `frame_id` (`Integer`) – the frame ID of the packet.
• `socket_id` (`Integer`) – the socket ID for which modification was requested.
• `option` (`SocketOption`) – the socket option of the parameter requested.
• `status` (`SocketStatus`) – the socket option status of the parameter requested.
• `option_data` (`Bytearray`, `optional`) – the option data. Optional.

See also:

XBeeAPIPacket
SocketOption
SocketStatus

Raises

• `ValueError` – if `frame_id` is less than 0 or greater than 255.
• **ValueError** – if `socket_id` is less than 0 or greater than 255.

```
static create_packet (raw, operating_mode)
```
Override method.

**Returns** `SocketOptionResponsePacket`.

**Raises**

- **InvalidPacketException** – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + frame id + socket id + option + status + checksum = 9 bytes).
- **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is not `ApiFrameType.SOCKET_OPTION_RESPONSE`.
- **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

- `XBeePacket.create_packet()`
- `XBeeAPIPacket._check_api_packet()`

```
needs_id()
```
Override method.

See also:

- `XBeeAPIPacket.needs_id()`

```
socket_id
```
Integer. Socket ID.

```
option
```
`SocketOption`. Socket option.

```
status
```
`SocketStatus`. Socket option status

```
option_data
```
Bytearray. Socket option data.

```
frame_id
```
Returns the frame ID of the packet.

**Returns** the frame ID of the packet.

**Return type** Integer
get_checksum()

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()

Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)

Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.socket.SocketConnectPacket (frame_id, socket_id, dest_port, dest_address_type, dest_address)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Connect packet. Packet is built using the parameters of the constructor.

Use this frame to create a socket connect message that causes the device to connect a socket to the given address and port.

For a UDP socket, this filters out any received responses that are not from the specified remote address and port.

Two frames occur in response:

• Socket Connect Response frame (SocketConnectResponsePacket): Arrives immediately and confirms the request.

• Socket Status frame (SocketStatePacket): Indicates if the connection was successful.

See also:

SocketConnectResponsePacket
SocketStatePacket
XBeeAPIPacket

Class constructor. Instantiates a new SocketConnectPacket object with the provided parameters.

Parameters

• frame_id (Integer) – the frame ID of the packet.

• socket_id (Integer) – the ID of the socket to connect.

• dest_port (Integer) – the destination port number.

• dest_address_type (Integer) – the destination address type. One of SocketConnectPacket.DEST_ADDRESS_BINARY or SocketConnectPacket.DEST_ADDRESS_STRING.

• dest_address (Bytearray or String) – the destination address.

See also:
SocketConnectPacket.DEST_ADDRESS_BINARY
SocketConnectPacket.DEST_ADDRESS_STRING
XBeeAPIPacket

Raises

- **ValueError** – if `frame_id` is less than 0 or greater than 255.
- **ValueError** – if `socket_id` is less than 0 or greater than 255.
- **ValueError** – if `dest_port` is less than 0 or greater than 65535.
- **ValueError** – if `dest_address_type` is different than `SocketConnectPacket.DEST_ADDRESS_BINARY` and `SocketConnectPacket.DEST_ADDRESS_STRING`.
- **ValueError** – if `dest_address` is `None` or does not follow the format specified in the configured type.

**DEST_ADDRESS_BINARY = 0**
Indicates the destination address field is a binary IPv4 address in network byte order.

**DEST_ADDRESS_STRING = 1**
Indicates the destination address field is a string containing either a dotted quad value or a domain name to be resolved.

`static create_packet(raw, operating_mode)`
Override method.

Returns `SocketConnectPacket`.

Raises

- **InvalidPacketException** – if the bytearray length is less than 11. (start delim. + length (2 bytes) + frame type + frame id + socket id + dest port (2 bytes) + dest address type + dest_address + checksum = 11 bytes).
- **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is not `ApiFrameType.SOCKET_CONNECT`.
- **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

```
XBeePacket.create_packet()
XBeeAPIPacket.__check_api_packet()
```

`needs_id()`
Override method.
XBee API Packet Documentation, Release 1.3.0

See also:

XBeeAPIPacket.needs_id()

socket_id
Integer. Socket ID.

dest_port
Integer. Destination port.

dest_address_type
Integer. Destination address type.

dest_address
Bytestring. Destination address.

frame_id
Returns the frame ID of the packet.

    Returns the frame ID of the packet.

    Return type Integer

generate_checksum()  
Returns the checksum value of this XBeePacket.

    The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

    Returns checksum value of this XBeePacket.

    Return type Integer

See also:

factory

generate_frame_spec_data()  
Override method.

See also:

XBeePacket.generate_frame_spec_data()

generate_frame_type()  
Returns the frame type of this packet.

    Returns the frame type of this packet.

    Return type ApiFrameType

See also:

ApiFrameType

2.5. API reference
get_frame_type_value()
Returns the frame type integer value of this packet.

  Returns the frame type integer value of this packet.
  Return type Integer

See also:

  ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

  Returns True if this packet is broadcast, False otherwise.
  Return type Boolean

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

  Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
  Returns raw bytearray of the XBeePacket.
  Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

  Returns dictionary with all information of the XBeePacket fields.
  Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

  Parameters data (Bytearray) – the bytearray to unescape.
  Returns data unescaped.
  Return type Bytearray

class digi.xbee.packets.socket.SocketConnectResponsePacket (frame_id, socket_id, status)
Bases: digi.xbee.packets.base.XBeeAPIPacket
This class represents a Socket Connect Response packet. Packet is built using the parameters of the constructor.

The device sends this frame in response to a Socket Connect (0x42) frame. The frame contains a status regarding the initiation of the connect.

See also:

  SocketConnectPacket
  XBeeAPIPacket

Class constructor. Instantiates a new SocketConnectPacket object with the provided parameters.

  Parameters
• **frame_id** (*Integer*) – the frame ID of the packet.
• **socket_id** (*Integer*) – the ID of the socket to connect.
• **status** (*SocketStatus*) – the socket connect status.

**See also:**

**XBeeAPIPacket**

**SocketStatus**

**Raises**

• **ValueError** – if `frame_id` is less than 0 or greater than 255.
• **ValueError** – if `socket_id` is less than 0 or greater than 255.

**static create_packet** (*raw, operating_mode*)

Override method.

**Returns** *SocketConnectResponsePacket*.

**Raises**

• **InvalidPacketException** – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + status + checksum = 8 bytes).
• **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
• **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See *SpecialByte*.
• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
• **InvalidPacketException** – if the frame type is not *ApiFrameType.SOCKET_CONNECT_RESPONSE*.
• **InvalidOperatingModeException** – if `operating_mode` is not supported.

**See also:**

`XBeePacket.create_packet()`

`XBeeAPIPacket._check_api_packet()`

**needs_id**()

Override method.

**See also:**

`XBeeAPIPacket.needs_id()`

**socket_id**

Integer. Socket ID.
status

SocketStatus. Socket connect status.

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType
is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output (escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data (data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.socket.SocketClosePacket (frame_id, socket_id)
Bases: digi.xbee.packets.base.XBeeAPIPacket
This class represents a Socket Close packet. Packet is built using the parameters of the constructor.
Use this frame to close a socket when given an identifier.

See also:

SocketCloseResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new SocketClosePacket object with the provided parameters.

Parameters
• frame_id (Integer) – the frame ID of the packet.
• socket_id (Integer) – the ID of the socket to close.

See also:

XBeeAPIPacket

Raises
• ValueError – if frame_id is less than 0 or greater than 255.
• ValueError – if socket_id is less than 0 or greater than 255.
static create_packet (raw, operating_mode)

Override method.

Returns SocketClosePacket.

Raises

• InvalidPacketException – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + socket id + checksum = 7 bytes).

• InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.

• InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

• InvalidPacketException – if the frame type is not ApiFrameType. SOCKET_CLOSE.

• InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id

Integer. Socket ID.

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory
get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.
Returns  data unescaped.

Return type  bytearray

class  digi.xbee.packets.socket.SocketCloseResponsePacket (frame_id, socket_id, status)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Close Response packet. Packet is built using the parameters of the constructor. The device sends this frame in response to a Socket Close (0x43) frame. Since a close will always succeed for a socket that exists, the status can be only one of two values:

• Success.
• Bad socket ID.

See also:

SocketClosePacket
XBeeAPIPacket

Class constructor. Instantiates a new SocketCloseResponsePacket object with the provided parameters.

Parameters

• frame_id (Integer) – the frame ID of the packet.
• socket_id (Integer) – the ID of the socket to close.
• status (SocketStatus) – the socket close status.

See also:

XBeeAPIPacket
SocketStatus

Raises

• ValueError – if frame_id is less than 0 or greater than 255.
• ValueError – if socket_id is less than 0 or greater than 255.

static  create_packet (raw, operating_mode)

Override method.

Returns  SocketCloseResponsePacket.

Raises

• InvalidPacketException – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + status + checksum = 8 bytes).
• InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
• InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).

• **InvalidPacketException** – if the frame type is not ApiFrameType.

• **InvalidOperatingModeException** – if operating_mode is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

```python
needs_id()
```

Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

```python
socket_id
```

Integer. Socket ID.

```python
status
```

*SocketStatus.* Socket close status.

```python
frame_id
```

Returns the frame ID of the packet.

Returns the frame ID of the packet.

```python
get_checksum()
```

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

```python
get_frame_spec_data()
```

Override method.

See also:

```python
XBeePacket.get_frame_spec_data()
```
get_frame_type()  
Returns the frame type of this packet.

    Returns the frame type of this packet.
    Return type ApiFrameType

See also:

    ApiFrameType

get_frame_type_value()  
Returns the frame type integer value of this packet.

    Returns the frame type integer value of this packet.
    Return type Integer

See also:

    ApiFrameType

is_broadcast()  
Returns whether this packet is broadcast or not.

    Returns True if this packet is broadcast, False otherwise.
    Return type Boolean

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

    Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
    Returns raw bytearray of the XBeePacket.
    Return type Bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.

    Returns dictionary with all information of the XBeePacket fields.
    Return type Dictionary

static unescape_data(data)  
Un-escapes the provided bytearray data.

    Parameters data (Bytearray) – the bytearray to unescape.
    Returns data unescaped.
    Return type Bytearray

class digi.xbee.packets.socket.SocketSendPacket (frame_id, socket_id, payload=None)
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Send packet. Packet is built using the parameters of the constructor.

A Socket Send message causes the device to transmit data using the current connection. For a nonzero frame ID, this will elicit a Transmit (TX) Status - 0x89 frame (TransmitStatusPacket).
This frame requires a successful Socket Connect - 0x42 frame first (SocketConnectPacket). For a socket that is not connected, the device responds with a Transmit (TX) Status - 0x89 frame with an error.

See also:

TransmitStatusPacket
XBeeAPIPacket

Class constructor. Instantiates a new SocketSendPacket object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the socket identifier.
- **payload** (*Bytearray, optional*) – data that is sent.

Raises

- **ValueError** – if **frame_id** is less than 0 or greater than 255.
- **ValueError** – if **socket_id** is less than 0 or greater than 255.

See also:

XBeeAPIPacket

```python
static create_packet(raw, operating_mode)
```

Override method.

Returns **SocketSendPacket**.

Raises

- **InvalidPacketException** – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + socket ID + checksum = 7 bytes).
- **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is not ApiFrameType. SOCKET_SEND.
- **InvalidOperatingModeException** – if **operating_mode** is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id

Integer. Socket ID.

payload

Bytearray. Payload to send.

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.
Returns the frame type integer value of this packet.

**Return type** Integer

See also:

(ApiFrameType)

**is_broadcast()**
Returns whether this packet is broadcast or not.

**Returns** True if this packet is broadcast, False otherwise.

**Return type** Boolean

(output (escaped=False))
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

**Parameters** escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

**Returns** raw bytearray of the XBeePacket.

**Return type** Bytearray

(to_dict())
Returns a dictionary with all information of the XBeePacket fields.

**Returns** dictionary with all information of the XBeePacket fields.

**Return type** Dictionary

(static unescape_data (data))
Un-escapes the provided bytearray data.

**Parameters** data (Bytearray) – the bytearray to unescape.

**Returns** data unescaped.

**Return type** Bytearray

**class** digi.xbee.packets.socket.SocketSendToPacket (frame_id, socket_id, dest_address, dest_port, payload=None)

**Bases:** digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Send packet. Packet is built using the parameters of the constructor.

A Socket SendTo (Transmit Explicit Data) message causes the device to transmit data using an IPv4 address and port. For a non-zero frame ID, this will elicit a Transmit (TX) Status - 0x89 frame (TransmitStatusPacket).

If this frame is used with a TCP, SSL, or a connected UDP socket, the address and port fields are ignored.

See also:

(TransmitStatusPacket)

(XBeeAPIPacket)

Class constructor. Instantiates a new SocketSendToPacket object with the provided parameters.

**Parameters**
• **frame_id** (*Integer*) – the frame ID of the packet.
• **socket_id** (*Integer*) – the socket identifier.
• **dest_address** (*IPv4Address*) – IPv4 address of the destination device.
• **dest_port** (*Integer*) – destination port number.
• **payload** (*Bytearray, optional*) – data that is sent.

**Raises**

• **ValueError** – if `frame_id` is less than 0 or greater than 255.
• **ValueError** – if `socket_id` is less than 0 or greater than 255.
• **ValueError** – if `dest_port` is less than 0 or greater than 65535.

**See also:**

`XBeeAPIPacket`

**static create_packet**( *raw, operating_mode*)

Override method.

**Returns** `SocketSendToPacket`.

**Raises**

• **InvalidPacketException** – if the bytearray length is less than 14. (start delim. + length (2 bytes) + frame type + frame id + socket ID + dest address (4 bytes) + dest port (2 bytes) + transmit options + checksum = 14 bytes).
• **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
• **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
• **InvalidPacketException** – if the frame type is not `ApiFrameType.SOCKET_SENDTO`.
• **InvalidOperatingModeException** – if `operating_mode` is not supported.

**See also:**

`XBeePacket.create_packet()`

`XBeeAPIPacket._check_api_packet()`

`needs_id()`

Override method.

**See also:**

`XBeeAPIPacket.needs_id()`
socket_id
   Integer. Socket ID.

dest_address
   ipaddress.IPv4Address. IPv4 address of the destination device.

dest_port
   Integer. Destination port.

payload
   Bytarray. Payload to send.

frame_id
   Returns the frame ID of the packet.
   
   Returns the frame ID of the packet.
   
   Return type Integer

get_checksum()
   Returns the checksum value of this XBeePacket.
   
   The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
   
   Returns checksum value of this XBeePacket.
   
   Return type Integer
   
   See also:

   factory

get_frame_spec_data()
   Override method.
   
   See also:

   XBeePacket.get_frame_spec_data()

get_frame_type()
   Returns the frame type of this packet.
   
   Returns the frame type of this packet.
   
   Return type ApiFrameType
   
   See also:

   ApiFrameType

get_frame_type_value()
   Returns the frame type integer value of this packet.
   
   Returns the frame type integer value of this packet.
   
   Return type Integer
See also:

 ApiServiceType

**is_broadcast()**

Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

**Return type** Boolean

**output**(escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters **escaped** (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

**Return type** Bytearray

**to_dict()**

Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

**Return type** Dictionary

**static unescape_data**(data)

Un-escapes the provided bytearray data.

Parameters **data** (Bytearray) – the bytearray to unescape.

Returns data unescaped.

**Return type** Bytearray

**class** digi.xbee.packets.socket.SocketBindListenPacket (frame_id, socket_id, source_port)

**Bases:** digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Bind/Listen packet. Packet is built using the parameters of the constructor.

Opens a listener socket that listens for incoming connections.

When there is an incoming connection on the listener socket, a Socket New IPv4 Client - 0xCC frame (SocketNewIPv4ClientPacket) is sent, indicating the socket ID for the new connection along with the remote address information.

For a UDP socket, this frame binds the socket to a given port. A bound UDP socket can receive data with a Socket Receive From: IPv4 - 0xCE frame (SocketReceiveFromIPv4Packet).

See also:

SocketNewIPv4ClientPacket
SocketReceiveFromIPv4Packet
XBeeAPIPacket

Class constructor. Instantiates a new SocketBindListenPacket object with the provided parameters.

Parameters
• **frame_id** (*Integer*) – the frame ID of the packet.
• **socket_id** (*Integer*) – socket ID to listen on.
• **source_port** (*Integer*) – the port to listen on.

**Raises**

• **ValueError** – if **frame_id** is less than 0 or greater than 255.
• **ValueError** – if **socket_id** is less than 0 or greater than 255.
• **ValueError** – if **source_port** is less than 0 or greater than 65535.

See also:

*XBeeAPIPacket*

**static create_packet** (*raw, operating_mode*)

Override method.

**Returns** *SocketBindListenPacket*.

**Raises**

• **InvalidPacketException** – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + frame id + socket ID + source port (2 bytes) + checksum = 9 bytes).
• **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
• **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
• **InvalidPacketException** – if the frame type is not ApiFrameType. SOCKET_BIND.
• **InvalidOperatingModeException** – if **operating_mode** is not supported.

See also:

*XBeePacket.create_packet()*

*XBeeAPIPacket._check_api_packet()*

**needs_id()**

Override method.

See also:

*XBeeAPIPacket.needs_id()*
socket_id
Integer. Socket ID.

source_port
Integer. Source port.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType
is_broadcast()  
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output (escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data (data)  
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.socket.SocketListenResponsePacket (frame_id, socket_id, status)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Listen Response packet. Packet is built using the parameters of the constructor.

The device sends this frame in response to a Socket Bind/Listen (0x46) frame (SocketBindListenPacket).

See also:

SocketBindListenPacket
XBeeAPIPacket

Class constructor. Instantiates a new SocketListenResponsePacket object with the provided parameters.

Parameters

• frame_id (Integer) – the frame ID of the packet.
• socket_id (Integer) – socket ID.
• status (SocketStatus) – socket listen status.

Raises

• ValueError – if frame_id is less than 0 or greater than 255.
• ValueError – if socket_id is less than 0 or greater than 255.

See also:
XBeeAPIPacket

SocketStatus

**static create_packet**(raw, operating_mode)

Override method.

**Returns** SocketListenResponsePacket.

**Raises**

- `InvalidPacketException` – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket ID + status + checksum = 8 bytes).

- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.

- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).

- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_LISTEN_RESPONSE`.

- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

- `XBeePacket.create_packet()`
- `XBeeAPIPacket._check_api_packet()`

**needs_id()**

Override method.

See also:

- `XBeeAPIPacket.needs_id()`

**socket_id**

Integer. Socket ID.

**status**

`SocketStatus`. Socket listen status.

**frame_id**

Returns the frame ID of the packet.

**Returns** the frame ID of the packet.

**Return type** Integer

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
**Returns** checksum value of this XBeePacket.

**Return type** Integer

See also:

`factory`

**get_frame_spec_data()**
Override method.

See also:

`XBeePacket.get_frame_spec_data()`

**get_frame_type()**
Returns the frame type of this packet.

**Returns** the frame type of this packet.

**Return type** `ApiFrameType`

See also:

`ApiFrameType`

**get_frame_type_value()**
Returns the frame type integer value of this packet.

**Returns** the frame type integer value of this packet.

**Return type** Integer

See also:

`ApiFrameType`

**is_broadcast()**
Returns whether this packet is broadcast or not.

**Returns** `True` if this packet is broadcast, `False` otherwise.

**Return type** Boolean

**output** *(escaped=False)*
Returns the raw bytearray of this XBeePacket, ready to be sent by the serial port.

**Parameters** `escaped` *(Boolean)* – indicates if the raw bytearray will be escaped or not.

**Returns** raw bytearray of the XBeePacket.

**Return type** Bytearray
to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type bytearray
class digi.xbee.packets.socket.SocketNewIPv4ClientPacket(socket_id, client_socket_id, remote_address, remote_port)
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket New IPv4 Client packet. Packet is built using the parameters of the constructor.
XBee Cellular modem uses this frame when an incoming connection is accepted on a listener socket.
This frame contains the original listener’s socket ID and a new socket ID of the incoming connection, along with the connection’s remote address information.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new SocketNewIPv4ClientPacket object with the provided parameters.

Parameters

• socket_id (Integer) – the socket ID of the listener socket.
• client_socket_id (Integer) – the socket ID of the new connection.
• remote_address (IPv4Address) – the remote IPv4 address.
• remote_port (Integer) – the remote port number.

Raises

• ValueError – if socket_id is less than 0 or greater than 255.
• ValueError – if client_socket_id is less than 0 or greater than 255.
• ValueError – if remote_port is less than 0 or greater than 65535.

See also:

XBeeAPIPacket

static create_packet(raw, operating_mode)
Override method.

Returns SocketNewIPv4ClientPacket.
Raises

- **InvalidPacketException** – if the bytearray length is less than 13. (start delim. + length (2 bytes) + frame type + socket ID + client socket ID + remote address (4 bytes) + remote port (2 bytes) + checksum = 13 bytes).
- **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is not `ApiFrameType.SOCKET_NEW_IPV4_CLIENT`.
- **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

```python
needs_id()
```
Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

```python
socket_id
```
Integer. Socket ID.

```python
client_socket_id
```
Integer. Client socket ID.

```python
remote_address
```
`ipaddress.IPv4Address`. Remote IPv4 address.

```python
remote_port
```
Integer. Remote port.

```python
frame_id
```
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

```python
get_checksum()
```
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer
See also:

factory

get_frame_spec_data()  
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()  
Returns the frame type of this packet.

    Returns  the frame type of this packet.
    Return type  ApiFrameType

See also:

ApiFrameType

get_frame_type_value()  
Returns the frame type integer value of this packet.

    Returns  the frame type integer value of this packet.
    Return type  Integer

See also:

ApiFrameType

is_broadcast()  
Returns whether this packet is broadcast or not.

    Returns  True if this packet is broadcast, False otherwise.
    Return type  Boolean

output (escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

    Parameters  escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
    Returns  raw bytearray of the XBeePacket.
    Return type  Bytearray

to_dict ()  
Returns a dictionary with all information of the XBeePacket fields.

    Returns  dictionary with all information of the XBeePacket fields.
Return type  Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters  data (Bytearray) – the bytearray to unescape.

Returns  data unescaped.

Return type  Bytearray

class  digi.xbee.packets.socket.SocketReceivePacket(frame_id, socket_id, payload=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Receive packet. Packet is built using the parameters of the constructor.
XBee Cellular modem uses this frame when it receives RF data on the specified socket.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new SocketReceivePacket object with the provided parameters.

Parameters

•  frame_id (Integer) – the frame ID of the packet.
•  socket_id (Integer) – the ID of the socket the data has been received on.
•  payload (Bytearray, optional) – data that is received.

Raises

•  ValueError – if frame_id is less than 0 or greater than 255.
•  ValueError – if socket_id is less than 0 or greater than 255.

See also:

XBeeAPIPacket

static create_packet(raw, operating_mode)
Override method.

Returns  SocketReceivePacket.

Raises

•  InvalidPacketException – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + socket ID + checksum = 7 bytes).
•  InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
•  InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.
•  InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).
• `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_RECEIVE`.
• `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

`needs_id()`  
Override method.  
See also:

```python
XBeeAPIPacket.needs_id()
```

`socket_id`  
Integer. Socket ID.

`payload`  
Bytestring. Payload that was received.

`frame_id`  
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type `Integer`

`get_checksum()`  
Returns the checksum value of this `XBeePacket`.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this `XBeePacket`.

Return type `Integer`

See also:

`factory`

`get_frame_spec_data()`  
Override method.

See also:

```python
XBeePacket.get_frame_spec_data()
```

`get_frame_type()`  
Returns the frame type of this packet.
Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output (escaped=False)

Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()

Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)

Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.socket.SocketReceiveFromPacket(frame_id, socket_id, source_address, source_port, payload=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Socket Receive From packet. Packet is built using the parameters of the constructor.

XBee Cellular modem uses this frame when it receives RF data on the specified socket. The frame also contains addressing information about the source.
See also:

XBeeAPIPacket

Class constructor. Instantiates a new SocketReceiveFromPacket object with the provided parameters.

Parameters

- **frame_id** (**Integer**) – the frame ID of the packet.
- **socket_id** (**Integer**) – the ID of the socket the data has been received on.
- **source_address** (**IPv4Address**) – IPv4 address of the source device.
- **source_port** (**Integer**) – source port number.
- **payload** (**Bytearray, optional**) – data that is received.

Raises

- **ValueError** – if **frame_id** is less than 0 or greater than 255.
- **ValueError** – if **socket_id** is less than 0 or greater than 255.
- **ValueError** – if **source_port** is less than 0 or greater than 65535.

See also:

XBeeAPIPacket

**frame_id**

Returns the frame ID of the packet.

**Returns**  the frame ID of the packet.

**Return type**  Integer

**get_checksum()**

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

**Returns**  checksum value of this XBeePacket.

**Return type**  Integer

See also:

factory

**get_frame_spec_data()**

Override method.

See also:

XBeePacket.get_frame_spec_data()
get_frame_type()
Returns the frame type of this packet.

    Returns the frame type of this packet.
    
    Return type ApiFrameType

See also:

    ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

    Returns the frame type integer value of this packet.
    
    Return type Integer

See also:

    ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

    Returns True if this packet is broadcast, False otherwise.
    
    Return type Boolean

output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

    Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
    
    Returns raw bytearray of the XBeePacket.
    
    Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

    Returns dictionary with all information of the XBeePacket fields.
    
    Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

    Parameters data (Bytearray) – the bytearray to unescape.
    
    Returns data unescaped.
    
    Return type Bytearray

static create_packet(raw, operating_mode)
Override method.

    Returns SocketReceiveFromPacket.
    
    Raises
• **InvalidPacketException** – if the bytearray length is less than 13. (start delim. + length (2 bytes) + frame type + frame id + socket ID + source address (4 bytes) + source port (2 bytes) + status + checksum = 14 bytes).

• **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.

• **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).

• **InvalidPacketException** – if the frame type is not `ApiFrameType.SOCKET_RECEIVE_FROM`.

• **InvalidOperatingModeException** – if `operating_mode` is not supported.

See also:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

**needs_id()**
 Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

**socket_id**
 Integer. Socket ID.

**source_address**
 `ipaddress.IPv4Address`. IPv4 address of the source device.

**source_port**
 Integer. Source port.

**payload**
 Bytestring. Payload that has been received.

```python
class digi.xbee.packets.socket.SocketStatePacket(socket_id, state)
Bases: digi.xbee.packets.base.XBeeAPIPacket
```

This class represents a Socket State packet. Packet is built using the parameters of the constructor.

This frame is sent out the device’s serial port to indicate the state related to the socket.

See also:

```python
XBeeAPIPacket
```

Class constructor. Instantiates a new `SocketStatePacket` object with the provided parameters.
Parameters

- `socket_id` (*Integer*) – the socket identifier.
- `state` (*SocketState*) – socket status.

Raises `ValueError` – if `socket_id` is less than 0 or greater than 255.

See also:

SockeState
XBeeAPIPacket

`frame_id`

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type: `Integer`

`get_checksum()`

Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type: `Integer`

See also:

`factory`

`get_frame_spec_data()`

Override method.

See also:

`XBeePacket.get_frame_spec_data()`

`get_frame_type()`

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type: `ApiFrameType`

See also:

`ApiFrameType`

`get_frame_type_value()`

Returns the frame type integer value of this packet.
Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast() Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False) Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict() Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data) Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

static create_packet(raw, operating_mode) Override method.

Returns SocketStatePacket.

Raises

- InvalidPacketException – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + socket ID + state + checksum = 7 bytes).

- InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

- InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.

- InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

- InvalidPacketException – if the frame type is not ApiFrameType.SOCKE...
See also:

```
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

**needs_id()**
Override method.

See also:

```
XBeeAPIPacket.needs_id()
```

**socket_id**
Integer. Socket ID.

**state**
```
SocketState. Socket state.
```

digi.xbee.packets.wifi module

class digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket (source_address, rssi, receive_options, rf_data=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a IO data sample RX indicator (Wi-Fi) packet. Packet is built using the parameters of the constructor or providing a valid API payload.

When the module receives an IO sample frame from a remote device, it sends the sample out the UART or SPI using this frame type. Only modules running API mode will be able to receive IO samples.

Among received data, some options can also be received indicating transmission parameters.

See also:

```
XBeeAPIPacket
```

Class constructor. Instantiates a new `IODataSampleRxIndicatorWifiPacket` object with the provided parameters.

**Parameters**

- **source_address** (ipaddress.IPv4Address) – the 64-bit source address.
- **rssi** (Integer) – received signal strength indicator.
- **receive_options** (Integer) – bitfield indicating the receive options.
- **rf_data** (Bytearray, optional) – received RF data. Optional.

Raises **ValueError** – if rf_data is not None and it’s not valid for create an IOSample.
static create_packet (raw, operating_mode)

Override method.

Returns IODataSampleRxIndicatorWifiPacket.

Raises

• InvalidPacketException – if the bytearray length is less than 16. (start delim. + length (2 bytes) + frame type + source addr. (4 bytes) + rssi + receive options + rf data (5 bytes) + checksum = 16 bytes).

• InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.

• InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

• InvalidPacketException – if the frame type is not ApiFrameType.IO_DATA_SAMPLE_RX_INDICATOR_WIFI.

• InvalidOperatingModeException – if operating_mode is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

source_address

ipaddress.IPv4Address. IPv4 source address.

rssi

Integer. Received Signal Strength Indicator (RSSI) value.

receive_options

Integer. Receive options bitfield.
rf_data
    Bytearray. Received RF data.

io_sample
    IO sample corresponding to the data contained in the packet.
    Type IOSample

frame_id
    Returns the frame ID of the packet.
    Returns the frame ID of the packet.
    Return type Integer

get_checksum()
    Returns the checksum value of this XBeePacket.
    The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
    Returns checksum value of this XBeePacket.
    Return type Integer

See also:

factory

get_frame_spec_data()
    Override method.
    See also:

    XBeePacket.get_frame_spec_data()

get_frame_type()
    Returns the frame type of this packet.
    Returns the frame type of this packet.
    Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
    Returns the frame type integer value of this packet.
    Returns the frame type integer value of this packet.
    Return type Integer

See also:
ApiFrameType

is_broadcast()
  Returns whether this packet is broadcast or not.

  Returns True if this packet is broadcast, False otherwise.

  Return type Boolean

output (escaped=False)
  Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

  Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

  Returns raw bytearray of the XBeePacket.

  Return type bytearray

to_dict()
  Returns a dictionary with all information of the XBeePacket fields.

  Returns dictionary with all information of the XBeePacket fields.

  Return type Dictionary

static unescape_data(data)
  Un-escapes the provided bytearray data.

  Parameters data (Bytearray) – the bytearray to unescape.

  Returns data unescaped.

  Return type bytearray

class digi.xbee.packets.wifi.RemoteATCommandWifiPacket (frame_id, dest_address, transmit_options, command, parameter=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a remote AT command request (Wi-Fi) packet. Packet is built using the parameters of the constructor or providing a valid API payload.

Used to query or set module parameters on a remote device. For parameter changes on the remote device to take effect, changes must be applied, either by setting the apply changes options bit, or by sending an AC command to the remote node.

Remote command options are set as a bitfield.

If configured, command response is received as a RemoteATCommandResponseWifiPacket.

See also:

RemoteATCommandResponseWifiPacket
XBeeAPIPacket

Class constructor. Instantiates a new RemoteATCommandWifiPacket object with the provided parameters.

Parameters

  - frame_id (integer) – the frame ID of the packet.
• `dest_address` (ipaddress.IPv4Address) – the IPv4 address of the destination device.

• `transmit_options` (Integer) – bitfield of supported transmission options.

• `command` (String) – AT command to send.

• `parameter` (Bytearray, optional) – AT command parameter. Optional.

**Raises**

• `ValueError` – if `frame_id` is less than 0 or greater than 255.

• `ValueError` – if length of `command` is different than 2.

**See also:**

`ipaddress.IPv4Address`
`RemoteATCmdOptions`
`XBeeAPIPacket`

```python
static create_packet (raw, operating_mode)
```

*Override method.*

**Returns** `RemoteATCommandWifiPacket`

**Raises**

• `InvalidPacketException` – if the `frame_type` is not `ApiFrameType.REMOTE_AT_COMMAND_REQUEST_WIFI`.

• `InvalidPacketException` – if the `frame_id` is not supported.

• `InvalidPacketException` – if the `transmit_options` are not supported.

• `InvalidPacketException` – if the `command` is not supported.

• `InvalidPacketException` – if the `frame_type` is not supported.

• `InvalidPacketException` – if the `operating_mode` is not supported.

**See also:**

`XBeePacket.create_packet()`
`XBeeAPIPacket._check_api_packet()`

```python
needs_id()
```

*Override method.*

**See also:**

`XBeeAPIPacket.needs_id()`
**dest_address**
IPaddress.IPv4Address. IPv4 destination address.

**transmit_options**
Integer. Transmit options bitfield.

**command**
String. AT command.

**parameter**
Bytearray. AT command parameter.

**frame_id**
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

**get_checksum()**
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

**factory**

**get_frame_spec_data()**
Override method.

See also:

XBeePacket.get_frame_spec_data()

**get_frame_type()**
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

**get_frame_type_value()**
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer
See also:

ApiFrameType

is_broadcast()  
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False)  
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()  
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)  
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

class digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket (frame_id, source_address, command, response_status, comm_value=None)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a remote AT command response (Wi-Fi) packet. Packet is built using the parameters of the constructor or providing a valid API payload.

If a module receives a remote command response RF data frame in response to a Remote AT Command Request, the module will send a Remote AT Command Response message out the UART. Some commands may send back multiple frames for example, Node Discover (ND) command.

This packet is received in response of a RemoteATCommandPacket.

Response also includes an ATCommandStatus object with the status of the AT command.

See also:

RemoteATCommandWifiPacket
ATCommandStatus
XBeeAPIPacket
Class constructor. Instantiates a new `RemoteATCommandResponseWifiPacket` object with the provided parameters.

**Parameters**

- **frame_id** (*Integer*) – the frame ID of the packet.
- **source_address** (*ipaddress.IPv4Address*) – the IPv4 address of the source device.
- **command** (*String*) – the AT command of the packet. Must be a string.
- **response_status** (*ATCommandStatus*) – the status of the AT command.
- **comm_value** (*Bytearray, optional*) – the AT command response value.

**Raises**

- **ValueError** – if `frame_id` is less than 0 or greater than 255.
- **ValueError** – if length of `command` is different than 2.

**See also:**

`ATCommandStatus`
`ipaddress.IPv4Address`

**static create_packet** (*raw, operating_mode*)

Override method.

**Returns** `RemoteATCommandResponseWifiPacket`.

**Raises**

- **InvalidPacketException** – if the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + frame id + source addr. (8 bytes) + command (2 bytes) + receive options + checksum = 17 bytes).
- **InvalidPacketException** – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)
- **InvalidPacketException** – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- **InvalidPacketException** – if the calculated checksum is different than the checksum field value (last byte).
- **InvalidPacketException** – if the frame type is not `ApiFrameType.REMOTE_AT_COMMAND_RESPONSE_WIFI`.
- **InvalidOperatingModeException** – if `operating_mode` is not supported.

**See also:**

`XBeePacket.create_packet()`
`XBeeAPIPacket._check_api_packet()`
needs_id()
    Override method.
    See also:

    XBeeAPIPacket.needs_id()

frame_id
    Returns the frame ID of the packet.
    Returns the frame ID of the packet.
    Return type Integer

get_checksum()
    Returns the checksum value of this XBeePacket.
    The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.
    Returns checksum value of this XBeePacket.
    Return type Integer
    See also:

    factory

get_frame_spec_data()
    Override method.
    See also:

    XBeePacket.get_frame_spec_data()

get_frame_type()
    Returns the frame type of this packet.
    Returns the frame type of this packet.
    Return type ApiFrameType
    See also:

    ApiFrameType

get_frame_type_value()
    Returns the frame type integer value of this packet.
    Returns the frame type integer value of this packet.
    Return type Integer
    See also:
is_broadcast() Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.

Return type Boolean

output(escaped=False) Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped(Boolean) – indicates if the raw bytearray will be escaped or not.

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict() Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data) Un-escapes the provided bytearray data.

Parameters data(Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

source_address
ipv4address.IPv4Address. IPv4 source address.

creedument
String. AT command.

status
ATCommandStatus. AT command response status.

command_value
Bytearray. AT command value.

digi.xbee.packets.zigbee module

class digi.xbee.packets.zigbee.RegisterJoiningDevicePacket(frame_id, registrant_address, options, key)

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Register Joining Device packet. Packet is built using the parameters of the constructor or providing a valid API payload.

Use this frame to securely register a joining device to a trust center. Registration is the process by which a node is authorized to join the network using a preconfigured link key or installation code that is conveyed to the trust center out-of-band (using a physical interface and not over-the-air).

If registering a device with a centralized trust center (EO = 2), then the key entry will only persist for KT seconds before expiring.
Registering devices in a distributed trust center (EO = 0) is persistent and the key entry will never expire unless explicitly removed.

To remove a key entry on a distributed trust center, this frame should be issued with a null (None) key. In a centralized trust center you cannot use this method to explicitly remove the key entries.

See also:

XBeeAPILacket

Class constructor. Instantiates a new RegisterJoiningDevicePacket object with the provided parameters.

Parameters

• frame_id (integer) – the frame ID of the packet.

• registrant_address (XBee64BitAddress) – the 64-bit address of the destination device.

• options (RegisterKeyOptions) – the register options indicating the key source.

• key (ByteArray) – key of the device to register. Up to 16 bytes if entering a Link Key or up to 18 bytes (16-byte code + 2 byte CRC) if entering an Install Code.

Raises ValueError – if frame_id is less than 0 or greater than 255.

See also:

XBee64BitAddress
XBeeAPILacket
RegisterKeyOptions

static create_packet (raw, operating_mode)

Override method.

Returns RegisterJoiningDevicePacket.

Raises

• InvalidPacketException – if the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + frame id + 64-bit registrant addr. (8 bytes) + 16-bit registrant addr. (2 bytes) + options + checksum = 17 bytes).

• InvalidPacketException – if the length field of ‘raw’ is different than its real length. (length field: bytes 2 and 3)

• InvalidPacketException – if the first byte of ‘raw’ is not the header byte. See SpecialByte.

• InvalidPacketException – if the calculated checksum is different than the checksum field value (last byte).

• InvalidPacketException – if the frame type is not ApiFrameType. REGISTER_JOINING_DEVICE.

• InvalidOperatingModeException – if operating_mode is not supported.

See also:
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Overwrite method.
See also:

XBeeAPIPacket.needs_id()

registrant_address
XBee64BitAddress. Registrant 64-bit address.

options
RegisterKeyOptions. Register options.

key
Bytearray. Register key.

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.
See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:
 ApiFrameType

get_frame_type_value() 
Returns the frame type integer value of this packet.
Returns the frame type integer value of this packet.
Return type Integer
See also:

ApiFrameType

is_broadcast() 
Returns whether this packet is broadcast or not.
Returns True if this packet is broadcast, False otherwise.
Return type Boolean

output(escaped=False) 
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.
Parameters escaped(Boolean) – indicates if the raw bytearray will be escaped or not.
Returns raw bytearray of the XBeePacket.
Return type bytearray

to_dict() 
Returns a dictionary with all information of the XBeePacket fields.
Returns dictionary with all information of the XBeePacket fields.
Return type Dictionary

static unescape_data(data) 
Un-escapes the provided bytearray data.
Parameters data(Bytearray) – the bytearray to unescape.
Returns data unescaped.
Return type bytearray

class digi.xbee.packets.zigbee.RegisterDeviceStatusPacket(frame_id, status) 
Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Register Device Status packet. Packet is built using the parameters of the constructor or providing a valid API payload.
This frame is sent out of the UART of the trust center as a response to a 0x24 Register Device frame, indicating whether the registration was successful or not.
See also:

RegisterJoiningDevicePacket
XBeeAPIPacket

2.5. API reference
Class constructor. Instantiates a new `RegisterDeviceStatusPacket` object with the provided parameters.

**Parameters**

- **frame_id** (*integer*) – the frame ID of the packet.
- **status** (*ZigbeeRegisterStatus*) – status of the register device operation.

**Raises** `ValueError` – if `frame_id` is less than 0 or greater than 255.

See also:

`XBeeAPIPacket`

`ZigbeeRegisterStatus`

\[
\text{static create_packet} (raw, \text{operating_mode})
\]

Override method.

**Returns** `RegisterDeviceStatusPacket`.

**Raises**

- `InvalidPacketException` – if the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + frame id + status + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different than its real length. (length field: bytes 1 and 3)
- `InvalidPacketException` – if the first byte of ‘raw’ is not the header byte. See `SpecialByte`.
- `InvalidPacketException` – if the calculated checksum is different than the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.REGISTER_JOINING_DEVICE_STATUS`.
- `InvalidOperatingModeException` – if `operating_mode` is not supported.

See also:

`XBeePacket.create_packet()`

`XBeeAPIPacket._check_api_packet()`

\[
\text{needs_id} ()
\]

Override method.

See also:

`XBeeAPIPacket.needs_id()`

\[
\text{status}
\]

`ZigbeeRegisterStatus`. Register device status.
frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.
Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.
Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.
Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.
Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns True if this packet is broadcast, False otherwise.
Return type  Boolean

**output (escaped=False)**

Returns the raw bytearray of this XBeepacket, ready to be send by the serial port.

Parameters  escaped **(Boolean)** – indicates if the raw bytearray will be escaped or not.

Returns  raw bytearray of the XBeepacket.

Return type  Bytearray

**to_dict ()**

Returns a dictionary with all information of the XBeepacket fields.

Returns  dictionary with all information of the XBeepacket fields.

Return type  Dictionary

**static unescape_data (data)**

Un-escapes the provided bytearray data.

Parameters  data **(Bytearray)** – the bytearray to unescape.

Returns  data unescaped.

Return type  Bytearray

**class digi.xbee.packets.zigbee.RouteRecordIndicatorPacket (x64bit_addr, x16bit_addr, receive_options, hops=None)**

Bases: digi.xbee.packets.base.XBeeAPIPacket

This class represents a Zigbee Route Record Indicator packet. Packet is built using the parameters of the constructor or providing a valid API payload.

The route record indicator is received whenever a device sends a Zigbee route record command. This is used with many-to-one routing to create source routes for devices in a network.

Among received data, some options can also be received indicating transmission parameters.

See also:

*ReceiveOptions*

*XBeeAPIPacket*

Class constructor. Instantiates a new RouteRecordIndicatorPacket object with the provided parameters.

Parameters

* x64bit_addr **(XBee64BitAddress)** – The 64-bit source address.
* x16bit_addr **(XBee16BitAddress)** – The 16-bit source address.
* receive_options **(Integer)** – Bitfield indicating the receive options.
* hops **(List, optional, default='None')** – List of 16-bit address of intermediate hops in the source route (excluding source and destination).

See also:
static create_packet (raw, operating_mode)

Override method.

Returns RouteRecordIndicatorPacket.

Raises
- InvalidPacketException – If the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + 64bit addr. + 16bit addr. + Receive options + num of addrs + checksum = 17 bytes).
- InvalidPacketException – If the length field of raw is different from its real length. (length field: bytes 1 and 3)
- InvalidPacketException – If the first byte of ‘raw’ is not the header byte. See SpecialByte.
- InvalidPacketException – If the calculated checksum is different from the checksum field value (last byte).
- InvalidPacketException – If the frame type is not ApiFrameType. ROUTE_RECORD_INDICATOR.
- InvalidPacketException – If the number of hops does not match with the number of 16-bit addresses.
- InvalidOperatingModeException – If operating_mode is not supported.

See also:

XBeePacket.create_packet ()
XBeeAPIPacket._check_api_packet ()

needs_id ()

Override method.

See also:

XBeeAPIPacket.needs_id ()

is_broadcast ()

Override method.

See also:

XBeeAPIPacket.is_broadcast ()
frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket.

The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type ApiFrameType

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

output(escaped=False)
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

Parameters escaped (Boolean) – indicates if the raw bytearray will be escaped or not.
XBee Python Library Documentation, Release 1.3.0

Returns raw bytearray of the XBeePacket.

Return type Bytearray

to_dict()
Returns a dictionary with all information of the XBeePacket fields.

Returns dictionary with all information of the XBeePacket fields.

Return type Dictionary

static unescape_data(data)
Un-escapes the provided bytearray data.

Parameters data (Bytearray) – the bytearray to unescape.

Returns data unescaped.

Return type Bytearray

x64bit_source_addr
Returns the 64-bit source address.

Returns The 64-bit source address.

Return type XBee64BitAddress

See also:

XBee64BitAddress

x16bit_source_addr
Returns the 16-bit source address.

Returns The 16-bit source address.

Return type XBee16BitAddress

See also:

XBee16BitAddress

receive_options
Returns the receive options bitfield.

Returns The receive options bitfield.

Return type Integer

See also:

XBeeReceiveOptions

number_of_hops
Returns the number of intermediate hops in the source route (excluding source and destination).

Returns The number of addresses.
**Return type**  Integer

**hops**
Returns the list of intermediate hops starting from the closest to destination hop and finishing with the closest to the source (excluding source and destination).

**Returns**  The list of 16-bit addresses of intermediate hops.

**Return type**  List

See also:

*XBee16BitAddress*

digi.xbee.packets.factory module

digi.xbee.packets.factory.build_frame(packet_bytearray, operating_mode=<OperatingMode.API_MODE: (1, 'API mode')>)

Creates a packet from raw data.

**Parameters**

- **packet_bytearray** (Bytearray) – the raw data of the packet to build.
- **operating_mode** (OperatingMode) – the operating mode in which the raw data has been captured.

See also:

*OperatingMode*

digi.xbee.util package

Submodules

digi.xbee.util.utils module

digi.xbee.util.utils.is_bit_enabled(number, position)

Returns whether the bit located at *position* within *number* is enabled or not.

**Parameters**

- **number** (Integer) – the number to check if a bit is enabled.
- **position** (Integer) – the position of the bit to check if is enabled in *number*.

**Returns**  True if the bit located at *position* within *number* is enabled, False otherwise.

**Return type**  Boolean

digi.xbee.util.utils.get_int_from_byte(number, offset, length)

Reads an integer value from the given byte using the provided bit offset and length.

**Parameters**

- **number** (Integer) – Byte to read the integer from.
• **offset** (*Integer*) – Bit offset inside the byte to start reading (LSB = 0, MSB = 7).
• **length** (*Integer*) – Number of bits to read.

**Returns** The integer value read.

**Return type** *Integer*

**Raises** *ValueError* – If *number* is lower than 0 or higher than 255. If *offset* is lower than 0 or higher than 7. If *length* is lower than 0 or higher than 8. If *offset* + *length* is higher than 8.

digi.xbee.util.utils.**hex_string_to_bytes** (*hex_string*)

Converts a String (composed by hex. digits) into a bytearray with same digits.

**Parameters**

- **hex_string** (*String*) – String (made by hex. digits) with “0x” header or not.

**Returns** bytearray containing the numeric value of the hexadecimal digits.

**Return type** *Bytearray*

**Raises** *ValueError* – if invalid literal for int() with base 16 is provided.

**Example**

```python
>>> a = "0xFFFE"
>>> for i in hex_string_to_bytes(a): print(i)
255
254
>>> print(type(hex_string_to_bytes(a)))
<type 'bytearray'>

>>> b = "FFFE"
>>> for i in hex_string_to_bytes(b): print(i)
255
254
>>> print(type(hex_string_to_bytes(b)))
<type 'bytearray'>
```

digi.xbee.util.utils.**int_to_bytes** (*number*, *num_bytes=None*)

Converts the provided integer into a bytearray.

If *number* has less bytes than *num_bytes*, the resultant bytearray is filled with zeros (0x00) starting at the beginning.

If *number* has more bytes than *num_bytes*, the resultant bytearray is returned without changes.

**Parameters**

- **number** (*Integer*) – the number to convert to a bytearray.
- **num_bytes** (*Integer*) – the number of bytes that the resultant bytearray will have.

**Returns** the bytearray corresponding to the provided number.

**Return type** *Bytearray*

**Example**
digi.xbee.util.utils.length_to_int(byte_array)
Calculates the length value for the given length field of a packet. Length field are bytes 1 and 2 of any packet.

Parameters byte_array (Bytearray) – length field of a packet.

Returns the length value.

Return type Integer

Raises ValueError – if byte_array is not a valid length field (it has length distinct than 0).

Example

```python
>>> b = bytearray([13,14])
>>> c = length_to_int(b)
>>> print("0x%02X" % c)
0x1314
>>> print(c)
4884
```

digi.xbee.util.utils.bytes_to_int(byte_array)
Converts the provided bytearray in an Integer. This integer is result of concatenate all components of byte_array and convert that hex number to a decimal number.

Parameters byte_array (Bytearray) – bytearray to convert in integer.

Returns the integer corresponding to the provided bytearray.

Return type Integer

Example

```python
>>> x = bytearray([0xA,0x0A,0x0A]) #this is 0xA0A0A0
>>> print(bytes_to_int(x))
657930
>>> b = bytearray([0x0A,0xAA]) #this is 0xAAA
>>> print(bytes_to_int(b))
2730
```

digi.xbee.util.utils.ascii_to_int(ni)
Converts a bytearray containing the ASCII code of each number digit in an Integer. This integer is result of the number formed by all ASCII codes of the bytearray.

Example

```python
>>> x = bytearray( [0x31,0x30,0x30] ) #0x31 => ASCII code for number 1.
>>> print(ascii_to_int(x))
100
```
**digi.xbee.util.utils.int_to_ascii(number)**

Converts an integer number to a bytearray. Each element of the bytearray is the ASCII code that corresponds to the digit of its position.

**Parameters** number (Integer) – the number to convert to an ASCII bytearray.

**Returns** the bytearray containing the ASCII value of each digit of the number.

**Return type** Bytearray

**Example**

```python
>>> x = int_to_ascii(100)
>>> print(x)
100
>>> print([i for i in x])
[49, 48, 48]
```

**digi.xbee.util.utils.int_to_length(number)**

Converts an integer into a bytearray of 2 bytes corresponding to the length field of a packet. If this bytearray has length 1, a byte with value 0 is added at the beginning.

**Parameters** number (Integer) – the number to convert to a length field.

**Returns**

**Raises** ValueError – if number is less than 0 or greater than 0xFFFF.

**Example**

```python
>>> a = 0
>>> print(hex_to_string(int_to_length(a)))
00 00

>>> a = 8
>>> print(hex_to_string(int_to_length(a)))
00 08

>>> a = 200
>>> print(hex_to_string(int_to_length(a)))
00 C8

>>> a = 0xFF00
>>> print(hex_to_string(int_to_length(a)))
FF 00

>>> a = 0xFF
>>> print(hex_to_string(int_to_length(a)))
00 FF
```

**digi.xbee.util.utils.hex_to_string(byte_array, pretty=True)**

Returns the provided bytearray in a pretty string format. All bytes are separated by blank spaces and printed in hex format.

**Parameters**
• **byte_array** (*Bytearray*) – the bytearray to print in pretty string.

• **pretty** (*Boolean, optional*) – True for pretty string format, False for plain string format. Default to True.

**Returns** the bytearray formatted in a string format.

**Return type** *String*

digi.xbee.util.utils.doc_enum(*enum_class, descriptions=None*)

Returns a string with the description of each value of an enumeration.

**Parameters**

• **enum_class** (*Enumeration*) – the Enumeration to get its values documentation.

• **descriptions** (*dictionary*) – each enumeration’s item description. The key is the enumeration element name and the value is the description.

**Returns** the string listing all the enumeration values and their descriptions.

**Return type** *String*

digi.xbee.util.utils.enable_logger(*name, level=10*)

Enables a logger with the given name and level.

**Parameters**

• **name** (*String*) – name of the logger to enable.

• **level** (*Integer*) – logging level value.

Assigns a default formatter and a default handler (for console).

digi.xbee.util.utils.disable_logger(*name*)

Disables the logger with the give name.

**Parameters**

• **name** (*String*) – the name of the logger to disable.


digi.xbee.util.utils.deprecated(*version, details=None*)

Decorates a method to mark as deprecated. This adds a deprecation note to the method docstring and also raises a :class:`warning.DeprecationWarning`.

**Parameters**

• **version** (*String*) – Version that deprecates this feature.

• **details** (*String, optional, default=``None``*) – Extra details to be added to the method docstring and warning.

**Submodules**

digi.xbee.comm_interface module

class digi.xbee.comm_interface.XBeeCommunicationInterface

Bases: object

This class represents the way the communication with the local XBee is established.

**open()**

Establishes the underlying hardware communication interface.

Subclasses may throw specific exceptions to signal implementation specific errors.
close()
Terminates the underlying hardware communication interface.
Subclasses may throw specific exceptions to signal implementation specific hardware errors.

is_interface_open
Returns whether the underlying hardware communication interface is active or not.

    Returns  Boolean. True if the interface is active, False otherwise.

wait_for_frame(operating_mode)
Reads the next API frame packet.

This method blocks until:
• A complete frame is read, in which case returns it.
• The configured timeout goes by, in which case returns None.
• Another thread calls quit_reading, in which case returns None.

This method is not thread-safe, so no more than one thread should invoke it at the same time.
Subclasses may throw specific exceptions to signal implementation specific hardware errors.

    Parameters operating_mode (OperatingMode) – the operating mode of the XBee connected to this hardware interface. Note: if this parameter does not match the connected XBee configuration, the behavior is undefined.

    Returns  the read packet as bytearray if a packet is read, None otherwise.

    Return type  Bytearray

quit_reading()
Makes the thread (if any) blocking on wait_for_frame return.
If a thread was blocked on wait_for_frame, this method blocks (for a maximum of ‘timeout’ seconds) until the blocked thread is resumed.

write_frame(frame)
Writes an XBee frame to the underlying hardware interface.
Subclasses may throw specific exceptions to signal implementation specific hardware errors.

    Parameters frame (Bytearray) – The XBee API frame packet to write. If the bytearray does not correctly represent an XBee frame, the behaviour is undefined.

    Returns  read timeout in seconds.

    Return type  Integer

digi.xbee.devices module

class digi.xbee.devices.AbstractXBeeDevice (local_xbee_device=None, serial_port=None, sync_ops_timeout=4, comm_iface=None)
Bases: object

This class provides common functionality for all XBee devices.
Class constructor. Instantiates a new AbstractXBeeDevice object with the provided parameters.

Parameters
• **local_xbee_device** (*XBeeDevice*, optional) – only necessary if XBee device is remote. The local XBee device that will behave as connection interface to communicate with the remote XBee one.

• **serial_port** (*XBeeSerialPort*, optional) – only necessary if the XBee device is local. The serial port that will be used to communicate with this XBee.

• **(Integer, default** (*sync_ops_timeout*) – *AbstractXBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS*) – the timeout (in seconds) that will be applied for all synchronous operations.

• **comm_iface** (*XBeeCommunicationInterface*, optional) – only necessary if the XBee device is local. The hardware interface that will be used to communicate with this XBee.

See also:

```python
XBeeDevice
XBeeSerialPort
```

```python
LOG_PATTERN = '{comm_iface:s} - {event:s} - {opmode:s}: {content:s}'
Pattern used to log packet events.
```

```python
update_device_data_from(*device*)
Updates the current device reference with the data provided for the given device.
This is only for internal use.

Parameters

- **device** (*AbstractXBeeDevice*) – the XBee device to get the data from.

Returns

- **True** if the device data has been updated, **False** otherwise.

Return type

- **Boolean**
```

```python
get_parameter(*parameter*, *parameter_value=None*)
Returns the value of the provided parameter via an AT Command.

Parameters

- **parameter** (*String*) – parameter to get.

- **parameter_value** (*Bytearray, optional*) – The value of the parameter to execute (if any).

Returns

- **the parameter value**.

Return type

- **Bytearray**
```

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.
```

```python
set_parameter(*parameter*, *value*)
Sets the value of a parameter via an AT Command.
```
If you send parameter to a local XBee device, all changes will be applied automatically, except for non-volatile memory, in which case you will need to execute the parameter “WR” via `AbstractXBeeDevice.execute_command()` method, or `AbstractXBeeDevice.apply_changes()` method.

If you are sending parameters to a remote XBee device, the changes will be not applied automatically, unless the “apply_changes” flag is activated.

You can set this flag via the method `AbstractXBeeDevice.enable_apply_changes()`.

This flag only works for volatile memory, if you want to save changed parameters in non-volatile memory, even for remote devices, you must execute “WR” command by one of the 2 ways mentioned above.

**Parameters**

- `parameter (String)` – parameter to set.
- `value (Bytearray)` – value of the parameter.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `ValueError` – if `parameter` is `None` or `value` is `None`.

**execute_command**(parameter)

Executes the provided command.

**Parameters**

- `parameter (String)` – The name of the AT command to be executed.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**apply_changes**()

Applies changes via AC command.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**write_changes**()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

2.5. API reference
If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method \texttt{is_apply_configuration_changes_enabled()} to get its status and \texttt{enable_apply_configuration_changes()} to enable/disable the option. If it is disabled, method \texttt{apply_changes()} can be used in order to manually apply the changes.

\textbf{Raises}

- \texttt{TimeoutException} – if the response is not received before the read timeout expires.
- \texttt{XBeeException} – if the XBee device’s serial port is closed.
- \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \texttt{ATCommandException} – if the response is not as expected.

\texttt{reset()}
Performs a software reset on this XBee device and blocks until the process is completed.

\textbf{Raises}

- \texttt{TimeoutException} – if the response is not received before the read timeout expires.
- \texttt{XBeeException} – if the XBee device’s serial port is closed.
- \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \texttt{ATCommandException} – if the response is not as expected.

\texttt{read_device_info(init=True, fire_event=True)}
Updates all instance parameters reading them from the XBee device.

\textbf{Parameters}

- \texttt{init} (Boolean, optional, default=`True`) – If False only not initialized parameters are read, all if True.
- \texttt{fire_event} (Boolean, optional, default=`True`) – True to launch and update event if any parameter changed, False otherwise.

\textbf{Raises}

- \texttt{TimeoutException} – if the response is not received before the read timeout expires.
- \texttt{XBeeException} – if the XBee device’s serial port is closed.
- \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \texttt{ATCommandException} – if the response is not as expected.

\texttt{get_node_id()}
Returns the Node Identifier (NI) value of the XBee device.

\textbf{Returns} the Node Identifier (NI) of the XBee device.

\textbf{Return type} String

\texttt{set_node_id(node_id)}
Sets the Node Identifier (NI) value of the XBee device.

\textbf{Parameters} \texttt{node_id(String)} – the new Node Identifier (NI) of the XBee device.
Raises

- `ValueError` – if `node_id` is `None` or its length is greater than 20.
- `TimeoutException` – if the response is not received before the read timeout expires.

`get_hardware_version()`
Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type: `HardwareVersion`

See also:

`HardwareVersion`

`get_firmware_version()`
Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type: `Bytearray`

`get_protocol()`
Returns the current protocol of the XBee device.

Returns the current protocol of the XBee device.

Return type: `XBeeProtocol`

See also:

`XBeeProtocol`

`get_16bit_addr()`
Returns the 16-bit address of the XBee device.

Returns the 16-bit address of the XBee device.

Return type: `XBee16BitAddress`

See also:

`XBee16BitAddress`

`set_16bit_addr(value)`
Sets the 16-bit address of the XBee device.

Parameters `value` (`XBee16BitAddress`) – the new 16-bit address of the XBee device.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPE API. This method only checks the cached value of the operating mode.
• `ATCommandException` – if the response is not as expected.
• `OperationNotSupportedException` – if the current protocol is not 802.15.4.

`get_64bit_addr()`  
Returns the 64-bit address of the XBee device.

    Returns  the 64-bit address of the XBee device.
    Return type  XBee64BitAddress

See also: 

`XBee64BitAddress`

`get_role()`  
Gets the XBee role.

    Returns  the role of the XBee.
    Return type  digi.xbee.models.protocol.Role

See also: 

digi.xbee.models.protocol.Role

`get_current_frame_id()`  
Returns the last used frame ID.

    Returns  the last used frame ID.
    Return type  Integer

`enable_apply_changes(value)`  
Sets the apply_changes flag.

    Parameters  value (Boolean) – True to enable the apply changes flag, False to disable it.

`is_apply_changes_enabled()`  
Returns whether the apply_changes flag is enabled or not.

    Returns  True if the apply_changes flag is enabled, False otherwise.
    Return type  Boolean

`is_remote()`  
Determines whether the XBee device is remote or not.

    Returns  True if the XBee device is remote, False otherwise.
    Return type  Boolean

`set_sync_ops_timeout(sync_ops_timeout)`  
Sets the serial port read timeout.

    Parameters  sync_ops_timeout (Integer) – the read timeout, expressed in seconds.
get_sync_ops_timeout()
Returns the serial port read timeout.
  Returns  the serial port read timeout in seconds.
  Return type  Integer

get_dest_address()
Returns the 64-bit address of the XBee device that data will be reported to.
  Returns  the address.
  Return type  XBee64BitAddress
  Raises  TimeoutException – if the response is not received before the read timeout expires.

See also:

XBee64BitAddress

set_dest_address(addr)
Sets the 64-bit address of the XBee device that data will be reported to.
  Parameters addr (XBee64BitAddress or RemoteXBeeDevice) – the address itself or the remote XBee device that you want to set up its address as destination address.
  Raises
  •  TimeoutException – If the response is not received before the read timeout expires.
  •  XBeeException – If the XBee device’s serial port is closed.
  •  InvalidOperatingModeException – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  •  ATCommandException – If the response is not as expected.
  •  ValueError – If addr is None.

get_pan_id()
Returns the operating PAN ID of the XBee device.
  Returns  operating PAN ID of the XBee device.
  Return type  Bytearray
  Raises  TimeoutException – if the response is not received before the read timeout expires.

set_pan_id(value)
Sets the operating PAN ID of the XBee device.
  Parameters value (Bytearray) – the new operating PAN ID of the XBee device. Must have only 1 or 2 bytes.
  Raises  TimeoutException – if the response is not received before the read timeout expires.

get_power_level()
Returns the power level of the XBee device.
  Returns  the power level of the XBee device.
  Return type  PowerLevel
  Raises  TimeoutException – if the response is not received before the read timeout expires.
See also:

PowerLevel

```
set_power_level(power_level)
```

Sets the power level of the XBee device.

Parameters

- \texttt{power\_level} (*PowerLevel*) – the new power level of the XBee device.

Raises \texttt{TimeoutException} – if the response is not received before the read timeout expires.

See also

PowerLevel

```
set_io_configuration(io_line, io_mode)
```

Sets the configuration of the provided IO line.

Parameters

- \texttt{io\_line} (*IOLine*) – the IO line to configure.
- \texttt{io\_mode} (*IOMode*) – the IO mode to set to the IO line.

Raises

- \texttt{TimeoutException} – if the response is not received before the read timeout expires.
- \texttt{XBeeException} – if the XBee device’s serial port is closed.
- \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \texttt{ATCommandException} – if the response is not as expected.

See also

IOLine

IOMode

```
get_io_configuration(io_line)
```

Returns the configuration of the provided IO line.

Parameters

- \texttt{io\_line} (*IOLine*) – the io line to configure.

Returns

the IO mode of the IO line provided.

Return type *IOMode*

Raises

- \texttt{TimeoutException} – if the response is not received before the read timeout expires.
- \texttt{XBeeException} – if the XBee device’s serial port is closed.
- \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.
• **OperationNotSupportedException** – if the received data is not an IO mode.

**get_io_sampling_rate()**
Returns the IO sampling rate of the XBee device.

**Returns**  the IO sampling rate of XBee device.

**Return type**  Integer

**Raises**
• **TimeoutException** – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

**set_io_sampling_rate(rate)**
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

**Parameters**  rate (Integer) – the new IO sampling rate of the XBee device in seconds.

**Raises**
• **TimeoutException** – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

**read_io_sample()**
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

**Returns**  the IO sample read from the XBee device.

**Return type**  IOSample

**Raises**
• **TimeoutException** – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

See also:

IOSample
get_adc_value(io_line)
Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.
set_io_configuration() and IOMode.ADC.

Parameters io_line (IOLine) – the IO line to get its ADC value.

Returns the analog value corresponding to the provided IO line.

Return type: Integer

 Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the response does not contain the value for the given IO line.

See also:

IOLine

set_pwm_duty_cycle(io_line, cycle)
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

• io_line (IOLine) – the IO Line to be assigned.
• cycle (Integer) – duty cycle in % to be assigned. Must be between 0 and 100.

 Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• ValueError – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

IOLine
IOMode.PWM
**get_pwm_duty_cycle** (*io_line*)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters

- **io_line** (*IOLine*) – the IO line to get its PWM duty cycle.

Returns

- the PWM duty cycle of the given IO line or **None** if the response is empty.

Return type

- **Integer**

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **ValueError** – if the passed **IO_LINE** has no PWM capability.

See also:

- **IOLine**

**get_dio_value** (*io_line*)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

Parameters

- **io_line** (*IOLine*) – the DIO line to get its digital value.

Returns

- current value of the provided IO line.

Return type

- **IOValue**

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

See also:

- **IOLine**
- **IOValue**

**set_dio_value** (*io_line, io_value*)
Sets the digital value (high or low) to the provided IO line.

Parameters
**io_line (IOLine)** – the digital IO line to set its value.

**io_value (IOValue)** – the IO value to set to the IO line.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**See also:**

- IOLine
- IOValue

**set_dio_change_detection (io_lines_set)**

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A None set of lines disables this feature.

**Parameters**

- **io_lines_set** – set of IOLine.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**See also:**

- IOLine

**get_api_output_mode ()**

Deprecated since version 1.3: Use **get_api_output_mode_value ()**

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

**Returns**

- the API output mode of the XBee device.

**Return type**

- APIOutputMode

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

ATCommandException – if the response is not as expected.

See also:

APIOutputMode

def get_api_output_mode_value()
Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns the parameter value.

Return type Bytearray

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

def set_api_output_mode(api_output_mode)

Sets the API output mode of the XBee device.

Parameters api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the current protocol is ZigBee

See also:
APIOutputMode

set_api_output_mode_value(api_output_mode)
Sets the API output mode of the XBee.

Parameters api_output_mode (Integer) – new API output mode options. Calculate this value using the method digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value() with a set of digi.xbee.models.mode.APIOutputModeBit.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

enable_bluetooth()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the AbstractXBeeDevice.update_bluetooth_password() method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

disable_bluetooth()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
get_bluetooth_mac_addr()
Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

Returns  The Bluetooth MAC address.

Return type  String

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBe device’s serial port is closed.
- InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

update_bluetooth_password(new_password)
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

Parameters  new_password(String) – New Bluetooth password.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBe device’s serial port is closed.
- InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)
Performs a firmware update operation of the device.

Parameters
- xml_firmware_file(String) – path of the XML file that describes the firmware to upload.
- xbee_firmware_file(String, optional) – location of the XBee binary firmware file.
- bootloader_firmware_file(String, optional) – location of the bootloader binary firmware file.
- timeout(Integer, optional) – the maximum time to wait for target read operations during the update process.
- progress_callback(Function, optional) – function to execute to receive progress information. Receives two arguments:
  – The current update task as a String
  – The current update task percentage as an Integer

Raises
- XBeeException – if the device is not open.
- InvalidOperatingModeException – if the device operating mode is invalid.
**OperationNotSupportedException** – if the firmware update is not supported in the XBee device.

**FirmwareUpdateException** – if there is any error performing the firmware update.

### apply_profile(profile_path, progress_callback=None)
Applies the given XBee profile to the XBee device.

#### Parameters
- **profile_path** *(String)* – path of the XBee profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

#### Raises
- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **UpdateProfileException** – if there is any error applying the XBee profile.
- **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

### reachable
Returns whether the XBee is reachable.

#### Returns
- **True** if the device is reachable, **False** otherwise.

#### Return type
**Boolean**

### scan_counter
Returns the scan counter for this node.

#### Returns
The scan counter for this node.

#### Return type
**Integer**

### log
Logger. The XBee device logger.

### class digi.xbee.devices.XBeeDevice(port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)
Bases: digi.xbee.devices.AbstractXBeeDevice

This class represents a non-remote generic XBee device.

This class has fields that are events. It’s recommended to use only the append() and remove() method on them, or += and -= operators. If you do something more with them, it’s for your own risk.

Class constructor. Instantiates a new XBeeDevice with the provided parameters.

#### Parameters
• **port** (*Integer or String*) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.

• **baud_rate** (*Integer*) – the serial port baud rate.

• (*Integer, default* (*_sync_ops_timeout*) – *serial.EIGHTBITS*): comm port bitsize.

• (*Integer, default* – *serial.STOPBITS_ONE*): comm port stop bits.

• (*Character, default* (*parity*) – *serial.PARITY_NONE*): comm port parity.

• (*Integer, default* – *FlowControl.NONE*): comm port flow control.

• (*Integer, default* – 3): the read timeout (in seconds).

• **comm_iface** (*XBeeCommunicationInterface*) – the communication interface.

    **Raises** All exceptions raised by PySerial’s Serial class constructor.

See also:


**TIMEOUT_READ_PACKET** = 3

Timeout to read packets.

**classmethod create_xbee_device**(comm_port_data)

Creates and returns an *XBeeDevice* from data of the port to which is connected.

**Parameters**

• **comm_port_data** (*Dictionary*) – dictionary with all comm port data needed.

• **dictionary keys are** (*The*) –

  “baudRate” → Baud rate.
  “port” → Port number.
  “bitSize” → Bit size.
  “stopBits” → Stop bits.
  “parity” → Parity.
  “flowControl” → Flow control.
  “timeout” for → Timeout for synchronous operations (in seconds).

**Returns** the XBee device created.

**Return type** *XBeeDevice*

**Raises** *SerialException* – if the port you want to open does not exist or is already opened.

See also:
Parameters **force_settings** *(Boolean, optional)* – True to open the device ensuring/forcing that the specified serial settings are applied even if the current configuration is different, False to open the device with the current configuration. Default to False.

Raisess
- *TimeoutException* – if there is any problem with the communication.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *XBeeException* – if the XBee device is already open.

close()
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

`get_parameter (param, parameter_value=None)`
Override.
See also:

    AbstractXBeeDevice.get_parameter()

`set_parameter (param, value)`
Override.
See: `AbstractXBeeDevice.set_parameter()`

`send_data (remote_xbee_device, data, transmit_options=0)`
Blocking method. This method sends data to a remote XBee device synchronously.
This method will wait for the packet response.
The default timeout for this method is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters
- **remote_xbee_device** *(RemoteXBeeDevice)* – the remote XBee device to send data to.
- **data** *(String or Bytearray)* – the raw data to send.
- **transmit_options** *(Integer, optional)* – transmit options, bitfield of `TransmitOptions`. Default to `TransmitOptions.NONE.value`.

Returns *XBeePacket* the response.

Raisess
- *ValueError* – if `remote_xbee_device` is None.
- *TimeoutException* – if this method can’t read a response packet in `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS` seconds.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *TransmitException* – if the status of the response received is not OK.
- *XBeeException* – if the XBee device’s serial port is closed.

See also:
send_data_async (remote_xbee_device, data, transmit_options=0)
Non-blocking method. This method sends data to a remote XBee device.
This method won’t wait for the response.

Parameters
- remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to send data to.
- data (String or Bytearray) – the raw data to send.
- transmit_options (Integer, optional) – transmit options, bitfield of TransmitOptions. Default to TransmitOptions.NONE.value.

Raises
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – if the XBee device’s serial port is closed.

See also:

send_data_broadcast (data, transmit_options=0)
Sends the provided data to all the XBee nodes of the network (broadcast).
This method blocks till a success or error transmit status arrives or the configured receive timeout expires.
The received timeout is configured using the AbstractXBeeDevice.set_sync_ops_timeout() method and can be consulted with AbstractXBeeDevice.get_sync_ops_timeout() method.

Parameters
- data (String or Bytearray) – data to send.
- transmit_options (Integer, optional) – transmit options, bitfield of TransmitOptions. Default to TransmitOptions.NONE.value.

Raises
- TimeoutException – if this method can’t read a response packet in XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS seconds.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TransmitException – if the status of the response received is not OK.
- XBeeException – if the XBee device’s serial port is closed.

send_user_data_relay (local_interface, data)
Sends the given data to the given XBee local interface.

Parameters
- local_interface (XBeeLocalInterface) – Destination XBee local interface.
• **data** (*Bytearray*) – Data to send.

**Raises**

- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ValueError* – if *local_interface* is *None*.
- *XBeeException* – if there is any problem sending the User Data Relay.

See also:

*XBeeLocalInterface*

**send_bluetooth_data**(*data*)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

**Parameters**

- **data** (*Bytearray*) – Data to send.

**Raises**

- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *XBeeException* – if there is any problem sending the data.

See also:

*XBeeDevice.send_micropython_data()*
*XBeeDevice.send_user_data_relay()*

**send_micropython_data**(*data*)

Sends the given data to the MicroPython interface using a User Data Relay frame.

**Parameters**

- **data** (*Bytearray*) – Data to send.

**Raises**

- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *XBeeException* – if there is any problem sending the data.

See also:

*XBeeDevice.send_bluetooth_data()*
*XBeeDevice.send_user_data_relay()*

**read_data**(*timeout=None*)

Reads new data received by this XBee device.

If a *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing in that case a *TimeoutException*.
Parameters **timeout** (*Integer, optional*) – read timeout in seconds. If it’s **None**, this method is non-blocking and will return **None** if there is no data available.

Returns the read message or **None** if this XBee did not receive new data.

Return type **XBeeMessage**

Raises

- **ValueError** – if a timeout is specified and is less than 0.
- **TimeoutException** – if a timeout is specified and no data was received during that time.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – if the XBee device’s serial port is closed.

See also:

**XBeeMessage**

**read_data_from** (*remote_xbee_device*, *timeout=None*)

Reads new data received from the given remote XBee device.

If a **timeout** is specified, this method blocks until new data is received or the timeout expires, throwing in that case a **TimeoutException**.

Parameters

- **remote_xbee_device** (*RemoteXBeeDevice*) – the remote XBee device that sent the data.
- **timeout** (*Integer, optional*) – read timeout in seconds. If it’s **None**, this method is non-blocking and will return **None** if there is no data available.

Returns

the read message sent by **remote_xbee_device** or **None** if this XBee did not receive new data.

Return type **XBeeMessage**

Raises

- **ValueError** – if a timeout is specified and is less than 0.
- **TimeoutException** – if a timeout is specified and no data was received during that time.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – if the XBee device’s serial port is closed.

See also:

**XBeeMessage**

**RemoteXBeeDevice**
has_packets()
Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

Returns True if this XBee device’s queue has packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

has_explicit_packets()
Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

Returns True if this XBee device’s queue has explicit packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

flush_queues()
Flushes the packets queue.

reset()
Override method.

See also:

AbstractXBeeDevice.reset()

add_packet_received_callback(callback)
Adds a callback for the event digi.xbee.reader.PacketReceived.

Parameters callback (Function) – the callback. Receives two arguments.

- The received packet as a digi.xbee.packets.base.XBeeAPIPacket

add_data_received_callback(callback)
Adds a callback for the event digi.xbee.reader.DataReceived.

Parameters callback (Function) – the callback. Receives one argument.

- The data received as an digi.xbee.models.message.XBeeMessage

add_modem_status_received_callback(callback)
Adds a callback for the event digi.xbee.reader.ModemStatusReceived.

Parameters callback (Function) – the callback. Receives one argument.

- The modem status as a digi.xbee.models.status.ModemStatus

add_io_sample_received_callback(callback)
Adds a callback for the event digi.xbee.reader.IOSampleReceived.
Parameters `callback` *(Function)* – the callback. Receives three arguments.

- The received IO sample as an `digi.xbee.io.IOSample`
- The remote XBee device who has sent the packet as a `RemoteXBeeDevice`
- The time in which the packet was received as an Integer

`add_expl_data_received_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.ExplicitDataReceived`.

Parameters `callback` *(Function)* – the callback. Receives one argument.

- The explicit data received as a `digi.xbee.models.message.ExplicitXBeeMessage`

`add_user_data_relay_received_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.RelayDataReceived`.

Parameters `callback` *(Function)* – the callback. Receives one argument.

- The relay data as a `digi.xbee.models.message.UserDataRelayMessage`

`add_bluetooth_data_received_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.BluetoothDataReceived`.

Parameters `callback` *(Function)* – the callback. Receives one argument.

- The Bluetooth data as a Bytearray

`add_micropython_data_received_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.MicroPythonDataReceived`.

Parameters `callback` *(Function)* – the callback. Receives one argument.

- The MicroPython data as a Bytearray

`add_socket_state_received_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.SocketStateReceived`.

Parameters `callback` *(Function)* – the callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a `SocketState`

`add_socket_data_received_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.SocketDataReceived`.

Parameters `callback` *(Function)* – the callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray

`add_socket_data_received_from_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.SocketDataReceivedFrom`.

Parameters `callback` *(Function)* – the callback. Receives three arguments.

- The socket ID as an Integer.
- A pair *(host, port)* of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
- The data received as Bytearray
**del_packet_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.PacketReceived` event.

**del_data_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.DataReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.DataReceived` event.

**del_modem_status_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

**del_io_sample_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.IOSampleReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.IOSampleReceived` event.

**del_expl_data_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.ExplicitDataReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ExplicitDataReceived` event.

**del_user_data_relay_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.

**del_bluetooth_data_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

**del_micropython_data_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

- **Parameters**
  - `callback` *(Function)* – the callback to delete.
  - **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.
del_socket_state_received_callback(callback)
    Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

    **Parameters**
    - **callback (Function)** – the callback to delete.
    
    **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

del_socket_data_received_callback(callback)
    Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

    **Parameters**
    - **callback (Function)** – the callback to delete.
    
    **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.

del_socket_data_received_from_callback(callback)
    Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

    **Parameters**
    - **callback (Function)** – the callback to delete.
    
    **Raises** `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

get_xbee_device_callbacks()
    Returns this XBee internal callbacks for process received packets.

    This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

    **Returns** `PacketReceived`

is_open()
    Returns whether this XBee device is open or not.

    **Returns** Boolean. `True` if this XBee device is open, `False` otherwise.

is_remote()
    Override method.

    See also:

    `AbstractXBeeDevice.is_remote()`

get_network()
    Returns this XBee device’s current network.

    **Returns** `XBeeDevice.XBeeNetwork`

send_packet_sync_and_get_response(packet_to_send, timeout=None)
    Override method.

    See also:

    `AbstractXBeeDevice._send_packet_sync_and_get_response()`

send_packet(packet, sync=False)
    Override method.
See also:

AbstractXBeeDevice._send_packet()

get_next_frame_id()
Returns the next frame ID of the XBee device.

Returns
The next frame ID of the XBee device.

Return type
Integer

add_route_received_callback(callback)
Adds a callback for the event RouteReceived. This works for Zigbee and Digimesh devices.

Parameters
callback (Function) – the callback. Receives three arguments.

• source (XBeeDevice): The source node.

• destination (RemoteXBeeDevice): The destination node.

• hops (List): List of intermediate hops from closest to source to closest to destination (RemoteXBeeDevice).

See also:

XBeeDevice.del_route_received_callback()

del_route_received_callback(callback)
Deletes a callback for the callback list of RouteReceived event.

Parameters
callback (Function) – the callback to delete.

See also:

XBeeDevice.add_route_received_callback()

get_route_to_node(remote, timeout=10)
Gets the route from this XBee to the given remote node.

Parameters

• remote (RemoteXBeeDevice) – The remote node.

• timeout (Float, optional, default=10) – Maximum number of seconds to wait for the route.

Returns

Tuple containing route data (None if the route was not read in the provided timeout):
- source (RemoteXBeeDevice): The source node of the route. - destination (RemoteXBeeDevice): The destination node of the route.

• hops (List): List of intermediate nodes (RemoteXBeeDevice) ordered from closest to source to closest to destination node (source and destination not included).
Return type: Tuple

**comm_iface**

*XBeeCommunicationInterface*. The hardware interface associated to the XBee device.

**serial_port**

*XBeeSerialPort*. The serial port associated to the XBee device.

**operating_mode**

*OperatingMode*. The operating mode of the XBee device.

**apply_changes()**
Applies changes via AC command.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**apply_profile** *(profile_path, progress_callback=None)*
Applies the given XBee profile to the XBee device.

Parameters

- **profile_path** *(String)* – path of the XBee profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – if the device is not open.
- `InvalidOperatingModeException` – if the device operating mode is invalid.
- `UpdateProfileException` – if there is any error applying the XBee profile.
- `OperationNotSupportedException` – if XBee profiles are not supported in the XBee device.

**disable_bluetooth()**
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**enable_apply_changes** *(value)*
Sets the apply_changes flag.
Parameters **value** (*Boolean*) – True to enable the apply changes flag, False to disable it.

**enable_bluetooth()**

Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**execute_command**(parameter)

Executes the provided command.

**Parameters** parameter (*String*) – The name of the AT command to be executed.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**get_16bit_addr()**

Returns the 16-bit address of the XBee device.

**Returns** the 16-bit address of the XBee device.

**Return type** `XBee16BitAddress`

See also: `XBee16BitAddress`

**get_64bit_addr()**

Returns the 64-bit address of the XBee device.

**Returns** the 64-bit address of the XBee device.

**Return type** `XBee64BitAddress`

See also: `XBee64BitAddress`

**get_adc_value**(io_line)

Returns the analog value of the provided IO line.
The provided IO line must be previously configured as ADC. To do so, use `AbstractXBeeDevice.set_io_configuration()` and `IOMode.ADC`.

**Parameters**

- **io_line** (IOLine) – the IO line to get its ADC value.

**Returns**

- the analog value corresponding to the provided IO line.

**Return type**

- Integer

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

**See also:**

- IOLine

---

**get_api_output_mode()**

Deprecated since version 1.3: Use `get_api_output_mode_value()`

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

**Returns**

- the API output mode of the XBee device.

**Return type**

- `APIOutputMode`

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**See also:**

- `APIOutputMode`

---

**get_api_output_mode_value()**

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

**Returns**

- the parameter value.
Return type  Bytearray

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()
Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

Returns  The Bluetooth MAC address.

Return type  String

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

get_current_frame_id()
Returns the last used frame ID.

Returns  the last used frame ID.

Return type  Integer

get_dest_address()
Returns the 64-bit address of the XBee device that data will be reported to.

Returns  the address.

Return type  XBee64BitAddress

Raises  `TimeoutException` – if the response is not received before the read timeout expires.

See also:

XBee64BitAddress
get_dio_value(\textit{io\_line})
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use \texttt{AbstractXBeeDevice.set\_io\_configuration()}.

Parameters \textbf{io\_line} (\texttt{IOLine}) – the DIO line to get its digital value.

Returns current value of the provided IO line.

Return type \texttt{IOValue}

Raises

- \texttt{TimeoutException} – if the response is not received before the read timeout expires.
- \texttt{XBeeException} – if the XBee device’s serial port is closed.
- \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \texttt{ATCommandException} – if the response is not as expected.
- \texttt{OperationNotSupportedException} – if the response does not contain the value for the given IO line.

See also:

\texttt{IOLine}
\texttt{IOValue}

get_firmware_version()
Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type \texttt{Bytearray}

get_hardware_version()
Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type \texttt{HardwareVersion}

See also:

\texttt{HardwareVersion}

get_io_configuration(\textit{io\_line})
Returns the configuration of the provided IO line.

Parameters \textbf{io\_line} (\texttt{IOLine}) – the IO line to configure.

Returns the IO mode of the IO line provided.

Return type \texttt{IOMode}

Raises

- \texttt{TimeoutException} – if the response is not received before the read timeout expires.
• ** XBeeException** – if the XBee device’s serial port is closed.

• ** InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ** ATCommandException** – if the response is not as expected.

• ** OperationNotSupportedException** – if the received data is not an IO mode.

### get_io_sampling_rate()

Returns the IO sampling rate of the XBee device.

**Returns**  the IO sampling rate of XBee device.

**Return type**  Integer

**Raises**

• ** TimeoutException** – if the response is not received before the read timeout expires.

• ** XBeeException** – if the XBee device’s serial port is closed.

• ** InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ** ATCommandException** – if the response is not as expected.

### get_node_id()

Returns the Node Identifier (NI) value of the XBee device.

**Returns**  the Node Identifier (NI) of the XBee device.

**Return type**  String

### get_pan_id()

Returns the operating PAN ID of the XBee device.

**Returns**  operating PAN ID of the XBee device.

**Return type**  Bytearray

**Raises**  **TimeoutException** – if the response is not received before the read timeout expires.

### get_power_level()

Returns the power level of the XBee device.

**Returns**  the power level of the XBee device.

**Return type**  **PowerLevel**

**Raises**  **TimeoutException** – if the response is not received before the read timeout expires.

**See also:**

*PowerLevel*

### get_protocol()

Returns the current protocol of the XBee device.

**Returns**  the current protocol of the XBee device.

**Return type**  **XBeeProtocol**

**See also:**
XBeeProtocol

**get_pwm_duty_cycle**(*io_line*)

Returns the PWM duty cycle in % corresponding to the provided IO line.

**Parameters**

- *io_line* *(IOLine)* – the IO line to get its PWM duty cycle.

**Returns**

the PWM duty cycle of the given IO line or `None` if the response is empty.

**Return type** *Integer*

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `ValueError` – if the passed `IO_LINE` has no PWM capability.

See also:

*IOLine*

**get_role**()

Gets the XBee role.

**Returns**

the role of the XBee.

**Return type** *digi.xbee.models.protocol.Role*

See also:

*digi.xbee.models.protocol.Role*

**get_sync_ops_timeout**()

Returns the serial port read timeout.

**Returns**

the serial port read timeout in seconds.

**Return type** *Integer*

**is_apply_changes_enabled**()

Returns whether the apply_changes flag is enabled or not.

**Returns**

`True` if the apply_changes flag is enabled, `False` otherwise.

**Return type** *Boolean*

**log**

Returns the XBee device log.

**Returns**

the XBee device logger.

**Return type** *Logger*
reachable
Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

read_device_info(init=True, fire_event=True)
Updates all instance parameters reading them from the XBee device.

Parameters

- init (Boolean, optional, default=`True`) – If False only not initialized parameters are read, all if True.
- fire_event (Boolean, optional, default=`True`) – True to launch and update event if any parameter changed, False otherwise.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

read_io_sample()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type IOSample

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

IOSample

scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

set_16bit_addr(value)
Sets the 16-bit address of the XBee device.

Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.
XBee Python Library Documentation, Release 1.3.0

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the current protocol is not 802.15.4.

`set_api_output_mode`(api_output_mode)

Deprecated since version 1.3: Use `set_api_output_mode_value()`

Sets the API output mode of the XBees device.

Parameters `api_output_mode` (`APIOutputMode`) – the new API output mode of the XBees device.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the current protocol is ZigBee

See also:

APIOutputMode

`set_api_output_mode_value`(api_output_mode)

Sets the API output mode of the XBees.

Parameters `api_output_mode` (`Integer`) – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:
digi.xbee.models.mode.APIOutputModeBit

**set_dest_address**(addr)
Sets the 64-bit address of the XBee device that data will be reported to.

Parameters **addr** ([XBee64BitAddress](https://example.com) or [RemoteXBeeDevice](https://example.com)) – the address itself or the remote XBee device that you want to set up its address as destination address.

Raises
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If the response is not as expected.
- `ValueError` – If `addr` is `None`.

**set_dio_change_detection**(io_lines_set)
Sets the digital IO lines to be monitored and sampled whenever their status changes.

A `None` set of lines disables this feature.

Parameters **io_lines_set** – set of [IOLine](https://example.com).

Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

[IOLine](https://example.com)

**set_dio_value**(io_line, io_value)
Sets the digital value (high or low) to the provided IO line.

Parameters
- **io_line** ([IOLine](https://example.com)) – the digital IO line to sets its value.
- **io_value** ([IOValue](https://example.com)) – the IO value to set to the IO line.

Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
set_io_configuration(io_line, io_mode)
Sets the configuration of the provided IO line.

Parameters
- **io_line** ([IOLine](# IOLine)) – the IO line to configure.
- **io_mode** ([IOMode](# IOMode)) – the IO mode to set to the IO line.

Raises
- [TimeoutException](# TimeoutException) – if the response is not received before the read timeout expires.
- [XBeeException](# XBeeException) – if the XBee device’s serial port is closed.
- [InvalidOperatingModeException](# InvalidOperatingModeException) – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- [ATCommandException](# ATCommandException) – if the response is not as expected.

See also:

[IOLine](# IOLine)
[IOMode](# IOMode)

set_io_sampling_rate(rate)
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** ([Integer](# Integer)) – the new IO sampling rate of the XBee device in seconds.

Raises
- [TimeoutException](# TimeoutException) – if the response is not received before the read timeout expires.
- [XBeeException](# XBeeException) – if the XBee device’s serial port is closed.
- [InvalidOperatingModeException](# InvalidOperatingModeException) – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- [ATCommandException](# ATCommandException) – if the response is not as expected.

set_node_id(node_id)
Sets the Node Identifier (NI) value of the XBee device.

Parameters **node_id** ([String](# String)) – the new Node Identifier (NI) of the XBee device.

Raises
- [ValueError](# ValueError) – if node_id is None or its length is greater than 20.
- [TimeoutException](# TimeoutException) – if the response is not received before the read timeout expires.

set_pan_id(value)
Sets the operating PAN ID of the XBee device.
Parameters `value` (*Bytearray*) – the new operating PAN ID of the XBee device. Must have only 1 or 2 bytes.

Raises `TimeoutException` – if the response is not received before the read timeout expires.

### set_power_level

Sets the power level of the XBee device.

**Parameters**

- `power_level` (*PowerLevel*) – the new power level of the XBee device.

Raises `TimeoutException` – if the response is not received before the read timeout expires.

See also:

- `PowerLevel`

### set_pwm_duty_cycle

Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

**Parameters**

- `io_line` (*IOLine*) – the IO Line to be assigned.
- `cycle` (*Integer*) – duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `ValueError` – if the given IO line does not have PWM capability or `cycle` is not between 0 and 100.

See also:

- `IOLine`
- `IOMode.PWM`

### set_sync_ops_timeout

Sets the serial port read timeout.

**Parameters**

- `sync_ops_timeout` (*Integer*) – the read timeout, expressed in seconds.

### update_bluetooth_password

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**

- `new_password` (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
XBee Python Library Documentation, Release 1.3.0

- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from(device)**

Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**
- `device (AbstractXBeeDevice)` – the XBee device to get the data from.

**Returns**
- `True` if the device data has been updated, `False` otherwise.

**Return type**
- Boolean

**update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)**

Performs a firmware update operation of the device.

**Parameters**
- `xml_firmware_file (String)` – path of the XML file that describes the firmware to upload.
- `xbee_firmware_file (String, optional)` – location of the XBee binary firmware file.
- `bootloader_firmware_file (String, optional)` – location of the bootloader binary firmware file.
- `timeout (Integer, optional)` – the maximum time to wait for target read operations during the update process.
- `progress_callback (Function, optional)` – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

**Raises**
- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.
- **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes()**

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method `is_apply_configuration_changes_enabled()` to get its status and
enable_apply_configuration_changes() to enable/disable the option. If it is disabled, method apply_changes() can be used in order to manually apply the changes.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

```python
class digi.xbee.devices.Raw802Device

Bases: digi.xbee.devices.XBeeDevice

This class represents a local 802.15.4 XBee device.

Class constructor. Instantiates a new Raw802Device with the provided parameters.

Parameters

- **port** (Integer or String) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.
- **baud_rate** (Integer) – the serial port baud rate.
- **flow_control** (serial.EIGHTBITS): comm port bit-size.
- **flow_control** (serial.STOPBITS_ONE): comm port stop bits.
- **parity** (serial.PARITY_NONE): comm port parity.
- **flow_control** (FlowControl.NONE): comm port flow control.
- **_sync_ops_timeout** (Integer, default: 3): the read timeout (in seconds).
- **comm_iface** (XBeeCommunicationInterface): the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor.: See also: XBeeDevice

XBeeDevice

XBeeDevice.__init__()

**open** (force_settings=False)

Override.

Raises

- **TimeoutException** – If there is any problem with the communication.
- **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **XBeeException** – If the protocol is invalid or if the XBee device is already open.

  See also:

  `XBeeDevice.open()`

  `get_protocol()`
  Override.

  See also:

  `XBeeDevice.get_protocol()`

  `get_ai_status()`
  Override.

  See also:

  `AbstractXBeeDevice._get_ai_status()`

  `send_data_64(x64addr, data, transmit_options=0)`
  Override.

  See also:

  `XBeeDevice.send_data_64()`

  `send_data_async_64(x64addr, data, transmit_options=0)`
  Override.

  See also:

  `XBeeDevice.send_data_async_64()`

  `send_data_16(x16addr, data, transmit_options=0)`
  Override.

  See also:

  `XBeeDevice._send_data_16()`

  `send_data_async_16(x16addr, data, transmit_options=0)`
  Override.

  See also:
XBeeDevice._send_data_async_16()

**add_bluetooth_data_received_callback** (*callback*)
Adds a callback for the event `digi.xbee.reader.BluetoothDataReceived`.

**Parameters**

- **callback** (*Function*) – the callback. Receives one argument.
  - The Bluetooth data as a Bytearray

**add_data_received_callback** (*callback*)
Adds a callback for the event `digi.xbee.reader.DataReceived`.

**Parameters**

- **callback** (*Function*) – the callback. Receives one argument.
  - The data received as a `digi.xbee.models.message.XBeeMessage`

**add_expl_data_received_callback** (*callback*)
Adds a callback for the event `digi.xbee.reader.ExplicitDataReceived`.

**Parameters**

- **callback** (*Function*) – the callback. Receives one argument.
  - The explicit data received as a `digi.xbee.models.message.ExplicitXBeeMessage`

**add_io_sample_received_callback** (*callback*)
Adds a callback for the event `digi.xbee.reader.IOSampleReceived`.

**Parameters**

- **callback** (*Function*) – the callback. Receives three arguments.
  - The received IO sample as a `digi.xbee.io.IOSample`
  - The remote XBee device who has sent the packet as a `RemoteXBeeDevice`
  - The time in which the packet was received as an Integer

**add_micropython_data_received_callback** (*callback*)
Adds a callback for the event `digi.xbee.reader.MicroPythonDataReceived`.

**Parameters**

- **callback** (*Function*) – the callback. Receives one argument.
  - The MicroPython data as a Bytearray

**add_modem_status_received_callback** (*callback*)
Adds a callback for the event `digi.xbee.reader.ModemStatusReceived`.

**Parameters**

- **callback** (*Function*) – the callback. Receives one argument.
  - The modem status as a `digi.xbee.models.status.ModemStatus`

**add_packet_received_callback** (*callback*)
Adds a callback for the event `digi.xbee.reader.PacketReceived`.

**Parameters**

- **callback** (*Function*) – the callback. Receives two arguments.
  - The received packet as a `digi.xbee.packets.base.XBeeAPIPacket`

**add_route_received_callback** (*callback*)
Adds a callback for the event `RouteReceived`. This works for Zigbee and Digimesh devices.

**Parameters**

- **callback** (*Function*) – the callback. Receives three arguments.
  - source (*XBeeDevice*): The source node.
  - destination (*RemoteXBeeDevice*): The destination node.
• hops (List): List of intermediate hops from closest to source to closest to destination (RemoteXBeeDevice).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback(callback)
Adds a callback for the event digi.xbee.reader.SocketDataReceived.

Parameters callback (Function) – the callback. Receives two arguments.
• The socket ID as an Integer.
• The data received as Bytearray

add_socket_data_received_from_callback(callback)
Adds a callback for the event digi.xbee.reader.SocketDataReceivedFrom.

Parameters callback (Function) – the callback. Receives three arguments.
• The socket ID as an Integer.
• A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
• The data received as Bytearray

add_socket_state_received_callback(callback)
Adds a callback for the event digi.xbee.reader.SocketStateReceived.

Parameters callback (Function) – the callback. Receives two arguments.
• The socket ID as an Integer.
• The state received as a SocketState

add_user_data_relay_received_callback(callback)
Adds a callback for the event digi.xbee.reader.RelayDataReceived.

Parameters callback (Function) – the callback. Receives one argument.
• The relay data as a digi.xbee.models.message.UserDataRelayMessage

apply_changes()
Applies changes via AC command.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

apply_profile(profile_path, progress_callback=None)
Applies the given XBee profile to the XBee device.

Parameters
• profile_path (String) – path of the XBee profile file to apply.
• **progress_callback** *(Function, optional)* –
  function to execute to receive progress information. Receives two arguments:
  – The current apply profile task as a String
  – The current apply profile task percentage as an Integer

**Raises**
  • **XBeeException** – if the device is not open.
  • **InvalidOperatingModeException** – if the device operating mode is invalid.
  • **UpdateProfileException** – if there is any error applying the XBee profile.
  • **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

**close()**
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

**comm_iface**
Returns the hardware interface associated to the XBee device.

  **Returns** the hardware interface associated to the XBee device.

  **Return type** XBeeCommunicationInterface

**See also:**

`XBeeSerialPort`

**classmethod create_xbee_device**(comm_port_data)
Creates and returns an **XBeeDevice** from data of the port to which is connected.

**Parameters**
  • **comm_port_data** *(Dictionary)* – dictionary with all comm port data needed.

  • **dictionary keys are** *(The)* –
    "baudRate" -> Baud rate.
    "port" -> Port number.
    "bitSize" -> Bit size.
    "stopBits" -> Stop bits.
    "parity" -> Parity.
    "flowControl" -> Flow control.
    "timeout" for -> Timeout for synchronous operations (in seconds).

  **Returns** the XBee device created.

  **Return type** XBeeDevice

  **Raises** **SerialException** – if the port you want to open does not exist or is already opened.

  **See also:**
XBeeDevice

del_bluetooth_data_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

del_data_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.DataReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.DataReceived` event.

del_expl_data_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.ExplicitDataReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.ExplicitDataReceived` event.

del_io_sample_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.IOSampleReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.IOSampleReceived` event.

del_micropython_data_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

del_modem_status_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

del_packet_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.PacketReceived` event.

del_route_received_callback(callback)
Deletes a callback for the callback list of `RouteReceived` event.

Parameters callback (Function) – the callback to delete.
See also:

```
XBeeDevice.add_route_received_callback()
```

### del_socket_data_received_callback(callback)

Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

- **Parameters**
  - `callback` *(Function)*: the callback to delete.
- **Raises**
  - `ValueError`: if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.

### del_socket_data_received_from_callback(callback)

Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

- **Parameters**
  - `callback` *(Function)*: the callback to delete.
- **Raises**
  - `ValueError`: if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

### del_socket_state_received_callback(callback)

Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

- **Parameters**
  - `callback` *(Function)*: the callback to delete.
- **Raises**
  - `ValueError`: if `callback` is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

### del_user_data_relay_received_callback(callback)

Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

- **Parameters**
  - `callback` *(Function)*: the callback to delete.
- **Raises**
  - `ValueError`: if `callback` is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.

### disable_bluetooth()

Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

- **Raises**
  - `TimeoutException`: if the response is not received before the read timeout expires.
  - `XBeeException`: if the XBee device’s serial port is closed.
  - `InvalidOperatingModeException`: if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

### enable_apply_changes(value)

Sets the apply_changes flag.

- **Parameters**
  - `value` *(Boolean)*: `True` to enable the apply changes flag, `False` to disable it.

### enable_bluetooth()

Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.
Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

`execute_command(parameter)`

Executes the provided command.

Parameters `parameter` (String) – The name of the AT command to be executed.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

`flush_queues()`

Flushes the packets queue.

`get_16bit_addr()`

Returns the 16-bit address of the XBee device.

Returns the 16-bit address of the XBee device.

Return type `XBee16BitAddress`

See also:

`XBee16BitAddress`

`get_64bit_addr()`

Returns the 64-bit address of the XBee device.

Returns the 64-bit address of the XBee device.

Return type `XBee64BitAddress`

See also:

`XBee64BitAddress`

`get_adc_value(io_line)`

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use `AbstractXBeeDevice.set_io_configuration()` and `IOMode.ADC`.

Parameters `io_line` (IOLine) – the IO line to get its ADC value.

Returns the analog value corresponding to the provided IO line.

Return type Integer
 Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

**IOLine**

get_api_output_mode()

Deprecated since version 1.3: Use `get_api_output_mode_value()`

Returns the API output mode of the XBees device.

The API output mode determines the format that the received data is output through the serial interface of the XBees device.

Returns: the API output mode of the XBees device.

Return type: `APIOutputMode`

 Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

**APIOutputMode**

get_api_output_mode_value()

Returns the API output mode of the XBees.

The API output mode determines the format that the received data is output through the serial interface of the XBees.

Returns: the parameter value.

Return type: `Bytearray`

 Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if it is not supported by the current protocol.

See also:

```python
digi.xbee.models.mode.APIOutputModeBit
```

### **get_bluetooth_mac_addr()**

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

**Returns**  The Bluetooth MAC address.

**Return type**  String

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

### **get_current_frame_id()**

Returns the last used frame ID.

**Returns**  the last used frame ID.

**Return type**  Integer

### **get_dest_address()**

Returns the 64-bit address of the XBee device that data will be reported to.

**Returns**  the address.

**Return type**  XBee64BitAddress

**Raises**  **TimeoutException** – if the response is not received before the read timeout expires.

See also:

```python
XBee64BitAddress
```

### **get_dio_value(io_line)**

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

**Parameters**  `io_line` ([IOLine](#)) – the DIO line to gets its digital value.

**Returns**  current value of the provided IO line.
Return type **IOValue**

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBees device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.
- *OperationNotSupportedException* – if the response does not contain the value for the given IO line.

See also:

**IOLine**

**IOValue**

---

### get_firmware_version()

Returns the firmware version of the XBees device.

**Returns** the hardware version of the XBees device.

**Return type** `Bytearray`

---

### get_hardware_version()

Returns the hardware version of the XBees device.

**Returns** the hardware version of the XBees device.

**Return type** `HardwareVersion`

See also:

**HardwareVersion**

---

### get_io_configuration(io_line)

Returns the configuration of the provided IO line.

**Parameters** `io_line` (**IOLine**) – the io line to configure.

**Returns** the IO mode of the IO line provided.

**Return type** `IOMode`

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBees device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.
- *OperationNotSupportedException* – if the received data is not an IO mode.
get_io_sampling_rate()

Returns the IO sampling rate of the XBee device.

Returns: the IO sampling rate of XBee device.

Return type: Integer

Raises:

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

get_network()

Returns this XBee device’s current network.

Returns: XBeeDevice.XBeeNetwork

get_next_frame_id()

Returns the next frame ID of the XBee device.

Returns: The next frame ID of the XBe device.

Return type: Integer

get_node_id()

Returns the Node Identifier (NI) value of the XBee device.

Returns: the Node Identifier (NI) of the XBe device.

Return type: String

get_pan_id()

Returns the operating PAN ID of the XBee device.

Returns: operating PAN ID of the XBe device.

Return type: bytearray

Raises: TimeoutException – if the response is not received before the read timeout expires.

get_parameter(param, parameter_value=None)

Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()

Returns the power level of the XBee device.

Returns: the power level of the XBe device.

Return type: PowerLevel

Raises: TimeoutException – if the response is not received before the read timeout expires.

See also:
PowerLevel

get_pwm_duty_cycle(io_line)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters io_line (IOLine) – the IO line to get its PWM duty cycle.
Returns the PWM duty cycle of the given IO line or None if the response is empty.
Return type Integer
Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not
  API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• ValueError – if the passed IO_LINE has no PWM capability.
See also:

IOLine

get_role()
Gets the XBee role.
Returns the role of the XBee.
Return type digi.xbee.models.protocol.Role
See also:

digi.xbee.models.protocol.Role

get_route_to_node (remote, timeout=10)
Gets the route from this XBee to the given remote node.
Parameters
• remote (RemoteXBeeDevice) – The remote node.
• timeout (Float, optional, default=10) – Maximum number of seconds to
  wait for the route.
Returns
Tuple containing route data (None if the route was not read in the provided timeout):
  - source (RemoteXBeeDevice): The source node of the route.
  - destination (RemoteXBeeDevice): The destination node
    of the route.
• hops (List): List of intermediate nodes (RemoteXBeeDevice) ordered from closest to source to closest to destination node (source and destination not included).

Return type  Tuple

get_sync_ops_timeout()  Returns the serial port read timeout.

Returns  the serial port read timeout in seconds.

Return type  Integer

get_xbee_device_callbacks()  Returns this XBee internal callbacks for process received packets.

This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

Returns  PacketReceived

has_explicit_packets()  Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

Returns  True if this XBee device’s queue has explicit packets, False otherwise.

Return type  Boolean

See also:

XBeeDevice.has_packets()

has_packets()  Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

Returns  True if this XBee device’s queue has packets, False otherwise.

Return type  Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()  Returns whether the apply_changes flag is enabled or not.

Returns  True if the apply_changes flag is enabled, False otherwise.

Return type  Boolean

is_open()  Returns whether this XBee device is open or not.

Returns  Boolean. True if this XBee device is open, False otherwise.

is_remote()  Override method.

See also:
AbstractXBeeDevice.is_remote()

log
Returns the XBee device log.

    Returns the XBee device logger.

    Return type Logger

operating_mode
Returns this XBee device’s operating mode.

    Returns OperatingMode. This XBee device’s operating mode.

reachable
Returns whether the XBee is reachable.

    Returns True if the device is reachable, False otherwise.

    Return type Boolean

read_data (timeout=None)
Reads new data received by this XBee device.

If a timeout is specified, this method blocks until new data is received or the timeout expires, throwing in that case a TimeoutException.

    Parameters timeout (Integer, optional) – read timeout in seconds. If it’s None, this method is non-blocking and will return None if there is no data available.

    Returns the read message or None if this XBee did not receive new data.

    Return type XBeeMessage

    Raises

    • ValueError – if a timeout is specified and is less than 0.

    • TimeoutException – if a timeout is specified and no data was received during that time.

    • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

    • XBeeException – if the XBee device’s serial port is closed.

See also:

XBeeMessage

read_data_from (remote_xbee_device, timeout=None)
Reads new data received from the given remote XBee device.

If a timeout is specified, this method blocks until new data is received or the timeout expires, throwing in that case a TimeoutException.

    Parameters

    • remote_xbee_device (RemoteXBeeDevice) – the remote XBee device that sent the data.
• **timeout** *(Integer, optional)—read timeout in seconds. If it’s None, this method is non-blocking and will return None if there is no data available.*

**Returns**

the read message sent by remote_xbee_device or None if this XBee did not receive new data.

**Return type** **XBeeMessage**

**Raises**

• **ValueError** – if a timeout is specified and is less than 0.

• **TimeoutException** – if a timeout is specified and no data was received during that time.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if the XBee device’s serial port is closed.

**See also:**

**XBeeMessage**

**RemoteXBeeDevice**

**read_device_info** *(init=True, fire_event=True)*

Updates all instance parameters reading them from the XBee device.

**Parameters**

• **init** *(Boolean, optional, default='True')—If False only not initialized parameters are read, all if True.*

• **fire_event** *(Boolean, optional, default='True')—True to launch and update event if any parameter changed, False otherwise.*

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

**read_io_sample** ()

Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

**Returns** the IO sample read from the XBee device.

**Return type** **IOSample**

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

See also:

`IOSample`

`reset()`

Override method.

See also:

AbstractXBeeDevice.reset()

`scan_counter`

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

`send_blueoth_data(data)`

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters `data` (Bytearray) – Data to send.

Raises

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `XBeeException` – if there is any problem sending the data.

See also:

XBeeDevice.send_micropython_data()
XBeeDevice.send_user_data_relay()

`send_data(remote_xbee_device, data, transmit_options=0)`

Blocking method. This method sends data to a remote XBee device synchronously.

This method will wait for the packet response.

The default timeout for this method is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

• `remote_xbee_device` (`RemoteXBeeDevice`) – the remote XBee device to send data to.

• `data` (String or Bytearray) – the raw data to send.

• `transmit_options` (Integer, optional) – transmit options, bitfield of `TransmitOptions`. Default to `TransmitOptions.NONE.value`. 
Returns `XBeePacket` the response.

Raises

- `ValueError` – if `remote_xbee_device` is `None`.
- `TimeoutException` – if this method can’t read a response packet in `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS` seconds.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – if the status of the response received is not OK.
- `XBeeException` – if the XBee device’s serial port is closed.

See also:

`RemoteXBeeDevice`

`XBeePacket`

`send_data_async (remote_xbee_device, data, transmit_options=0)`

Non-blocking method. This method sends data to a remote XBee device.

This method won’t wait for the response.

Parameters

- `remote_xbee_device` (`RemoteXBeeDevice`) – the remote XBee device to send data to.
- `data` (`String or Bytearray`) – the raw data to send.
- `transmit_options` (`Integer, optional`) – transmit options, bitfield of `TransmitOptions`. Default to `TransmitOptions.NONE.value`.

Raises

- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – if the XBee device’s serial port is closed.

See also:

`RemoteXBeeDevice`

`send_data_broadcast (data, transmit_options=0)`

Sends the provided data to all the XBee nodes of the network (broadcast).

This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using the `AbstractXBeeDevice.set_sync_ops_timeout()` method and can be consulted with `AbstractXBeeDevice.get_sync_ops_timeout()` method.

Parameters

- `data` (`String or Bytearray`) – data to send.
• **transmit_options** (*Integer, optional*) – transmit options, bitfield of `TransmitOptions`. Default to `TransmitOptions.NONE.value`.

Raises

• **TimeoutException** – if this method can’t read a response packet in `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS` seconds.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **TransmitException** – if the status of the response received is not OK.

• **XBeeException** – if the XBee device’s serial port is closed.

`send_micropython_data` (*data*)

Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters

• **data** (*Bytearray*) – Data to send.

Raises

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if there is any problem sending the data.

See also:

- `XBeeDevice.send_bluetooth_data()`
- `XBeeDevice.send_user_data_relay()`

`send_packet` (*packet, sync=False*)

Override method.

See also:

- `AbstractXBeeDevice._send_packet()`

`send_packet_sync_and_get_response` (*packet_to_send, timeout=None*)

override method.

See also:

- `AbstractXBeeDevice._send_packet_sync_and_get_response()`

`send_user_data_relay` (*local_interface, data*)

Sends the given data to the given XBee local interface.

Parameters

• **local_interface** (*XBeeLocalInterface*) – Destination XBee local interface.

• **data** (*Bytearray*) – Data to send.

Raises
InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

ValueError – if local_interface is None.

XBeeException – if there is any problem sending the User Data Relay.

See also:

XBeeLocalInterface

 serial_port
 Returns the serial port associated to the XBee device, if any.

 Returns
 the serial port associated to the XBee device. Returns None if the local XBee does not use serial communication.

 Return type XBeeSerialPort
 See also:

XBeeSerialPort

 set_16bit_addr (value)
 Sets the 16-bit address of the XBee device.

 Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.

 Raises

 TimeoutException – if the response is not received before the read timeout expires.

 XBeeException – if the XBee device’s serial port is closed.

 InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

 ATCommandException – if the response is not as expected.

 OperationNotSupportedException – if the current protocol is not 802.15.4.

 set_api_output_mode (api_output_mode)
 Deprecated since version 1.3: Use set_api_output_mode_value()

 Sets the API output mode of the XBee device.

 Parameters api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

 Raises

 TimeoutException – if the response is not received before the read timeout expires.

 XBeeException – if the XBee device’s serial port is closed.

 InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

 ATCommandException – if the response is not as expected.
• **OperationNotSupportedException** – if the current protocol is ZigBee

See also:

`APIOutputMode`

`set_api_output_mode_value(api_output_mode)`

Sets the API output mode of the XBee.

**Parameters** `api_output_mode` (*Integer*) – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBees serial port is closed.
• **InvalidOperatingModeException** – if the XBees operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.
• **OperationNotSupportedException** – if it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

`set_dest_address(addr)`

Sets the 64-bit address of the XBee device that data will be reported to.

**Parameters** `addr` (*XBee64BitAddress* or *RemoteXBeeDevice*) – the address itself or the remote XBee device that you want to set up its address as destination address.

**Raises**

• **TimeoutException** – If the response is not received before the read timeout expires.
• **XBeeException** – If the XBees serial port is closed.
• **InvalidOperatingModeException** – If the XBees operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – If the response is not as expected.
• **ValueError** – If `addr` is None.

`set_dio_change_detection(io_lines_set)`

Sets the digital IO lines to be monitored and sampled whenever their status changes. A None set of lines disables this feature.

**Parameters** `io_lines_set` – set of `IOLine`.

**Raises**
• `TimeoutException` – if the response is not received before the read timeout expires.
• `XBeeException` – if the XBe device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ATCommandException` – if the response is not as expected.

See also:

```python
IOLine
```

`set_dio_value(io_line, io_value)`  
Sets the digital value (high or low) to the provided IO line.

Parameters

- `io_line` (`IOLine`) – the digital IO line to sets its value.
- `io_value` (`IOValue`) – the IO value to set to the IO line.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

```python
IOLine
IOValue
```

`set_io_configuration(io_line, io_mode)`  
Sets the configuration of the provided IO line.

Parameters

- `io_line` (`IOLine`) – the IO line to configure.
- `io_mode` (`IOMode`) – the IO mode to set to the IO line.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:
IOLine
IOMode

**set_io_sampling_rate** (*rate*)
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>rate</em></td>
<td>Integer</td>
<td>the new IO sampling rate of the XBee device in seconds.</td>
</tr>
</tbody>
</table>

**Raises**

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.

**set_node_id** (*node_id*)
Sets the Node Identifier (NI) value of the XBee device.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>node_id</em></td>
<td>String</td>
<td>the new Node Identifier (NI) of the XBee device.</td>
</tr>
</tbody>
</table>

**Raises**

- *ValueError* – if *node_id* is *None* or its length is greater than 20.
- *TimeoutException* – if the response is not received before the read timeout expires.

**set_pan_id** (*value*)
Sets the operating PAN ID of the XBee device.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>value</em></td>
<td>Bytearray</td>
<td>the new operating PAN ID of the XBee device.</td>
</tr>
</tbody>
</table>

**Raises** *TimeoutException* – if the response is not received before the read timeout expires.

**set_parameter** (*param*, *value*)
Override.

See: `AbstractXBeeDevice.set_parameter()`

**set_power_level** (*power_level*)
Sets the power level of the XBee device.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>power_level</em></td>
<td>PowerLevel</td>
<td>the new power level of the XBee device.</td>
</tr>
</tbody>
</table>

**Raises** *TimeoutException* – if the response is not received before the read timeout expires.

See also:

*PowerLevel*

**set_pwm_duty_cycle** (*io_line*, *cycle*)
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>io_line</em></td>
<td></td>
<td>the provided IO line.</td>
</tr>
<tr>
<td><em>cycle</em></td>
<td></td>
<td>the duty cycle in % of the provided IO line.</td>
</tr>
</tbody>
</table>
• **io_line** (*IOLine*) – the IO Line to be assigned.

• **cycle** (*Integer*) – duty cycle in % to be assigned. Must be between 0 and 100.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **ValueError** – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

**See also:**

*IOLine
IOMode.PWM*

**set_sync_ops_timeout** (*sync_ops_timeout*)

Sets the serial port read timeout.

**Parameters**

* sync_ops_timeout (*Integer*) – the read timeout, expressed in seconds.

**update_bluetooth_password** (*new_password*)

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**

* new_password (*String*) – New Bluetooth password.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from** (*device*)

Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**

* device (*AbstractXBeeDevice*) – the XBee device to get the data from.

**Returns**

* True if the device data has been updated, False otherwise.

**Return type**

* Boolean

**update_firmware** (*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the device.

**Parameters**

• *xml_firmware_file* (*String*) – path of the XML file that describes the firmware to upload.
• **xbee_firmware_file** *(String, optional)* – location of the XBee binary firmware file.

• **bootloader_firmware_file** *(String, optional)* – location of the bootloader binary firmware file.

• **timeout** *(Integer, optional)* – the maximum time to wait for target read operations during the update process.

• **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  
  – The current update task as a String
  – The current update task percentage as an Integer

**Raises**

• **XBeeException** – if the device is not open.

• **InvalidOperatingModeException** – if the device operating mode is invalid.

• **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.

• **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes()**

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method **is_apply_configuration_changes_enabled()** to get its status and **enable_apply_configuration_changes()** to enable/disable the option. If it is disabled, method **apply_changes()** can be used in order to manually apply the changes.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

```python
class digi.xbee.devices.DigiMeshDevice (port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)
```

**Bases:** `digi.xbee.devices.XBeeDevice`

This class represents a local DigiMesh XBee device.

Class constructor. Instantiates a new **DigiMeshDevice** with the provided parameters.
Parameters

- **port**: a string or integer specifying the serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.

- **baud_rate**: the serial port baud rate.

- **(Integer, default (flow_control) – serial.EIGHTBITS):** comm port bit-size.

- **(Integer, default (serial.STOPBITS_ONE):** comm port stop bits.

- **(Character, default (serial.PARITY_NONE):** comm port parity.

- **(Integer, default – FlowControl.NONE):** comm port flow control.

- **_sync_ops_timeout (Integer, default: 3):** the read timeout (in seconds).

- **comm_iface**: the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor.

See also:

**XBeeDevice**

XBeeDevice.__init__()

---

**open** *(force_settings=False)*

Override.

Raises

- **TimeoutException** – If there is any problem with the communication.

- **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **XBeeException** – If the protocol is invalid or if the XBee device is already open.

See also:

**XBeeDevice.open()**

---

**get_protocol** *

Override.

See also:

**XBeeDevice.get_protocol()**

---

**build_aggregate_routes** *

Automatically build routes to this node. All nodes in the network will build routes to this node. The receiving node establishes a route back to this node.

Raises
• **TimeoutException** – If the response is not received before the read timeout expires.
• **XBeeException** – If the XBee device’s serial port is closed.
• **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – If the response is not as expected.

**send_data_64** *(x64addr, data, transmit_options=0)*
Override.
See also:

```python
XBeeDevice.send_data_64()
```

**send_data_async_64** *(x64addr, data, transmit_options=0)*
Override.
See also:

```python
XBeeDevice.send_data_async_64()
```

**read_expl_data** *(timeout=None)*
Override.
See also:

```python
XBeeDevice.read_expl_data()
```

**read_expl_data_from** *(remote_xbee_device, timeout=None)*
Override.
See also:

```python
XBeeDevice.read_expl_data_from()
```

**send_expl_data** *(remote_xbee_device, data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)*
Override.
See also:

```python
XBeeDevice.send_expl_data()
```

**send_expl_data_broadcast** *(data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)*
Override.
See also:

XBeeDevice._send_expl_data_broadcast()

```python
send_expl_data_async(remote_xbee_device, data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)
```

Override.

See also:

XBeeDevice.send_expl_data_async()

```python
get_neighbors(neighbor_callback=None, process_finished_callback=None, timeout=None)
```

Returns the neighbors of this XBee. If `neighbor_callback` is not defined, the process blocks during the specified timeout.

**Parameters**

- `neighbor_callback` *(Function, optional, default='None'* – method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
- `process_finished_callback` *(Function, optional, default='None'* – method to execute when the process finishes. Receives two arguments:
  - The XBee device that is searching for its neighbors.
  - A list with the discovered neighbors.
  - An error message if something went wrong.
- `timeout` *(Float, optional, default=``NeighborFinder.DEFAULT_TIMEOUT``)* – The timeout in seconds.

**Returns**

List of `com.digi.models.zdo.Neighbor` when `neighbor_callback` is defined, `None` otherwise (in this case neighbors are received in the callback).

**Return type** List

**Raises** `OperationNotSupportedException` – If XBee protocol is not DigiMesh.

See also:

```python
com.digi.models.zdo.Neighbor
```

```python
add_bluetooth_data_received_callback(callback)
```

Adds a callback for the event `digi.xbee.reader.BluetoothDataReceived`.

**Parameters** `callback` *(Function)* – the callback. Receives one argument.

- The Bluetooth data as a Bytearray
add_data_received_callback(callback)
Adds a callback for the event \texttt{digi.xbee.reader.DataReceived}.

\textbf{Parameters} \texttt{callback (Function)} – the callback. Receives one argument.

- The data received as an \texttt{digi.xbee.models.message.XBeeMessage}

add_expl_data_received_callback(callback)
Adds a callback for the event \texttt{digi.xbee.reader.ExplicitDataReceived}.

\textbf{Parameters} \texttt{callback (Function)} – the callback. Receives one argument.

- The explicit data received as a \texttt{digi.xbee.models.message.ExplicitXBeeMessage}

add_io_sample_received_callback(callback)
Adds a callback for the event \texttt{digi.xbee.reader.IOSampleReceived}.

\textbf{Parameters} \texttt{callback (Function)} – the callback. Receives three arguments.

- The received IO sample as an \texttt{digi.xbee.io.IOSample}
- The remote XBee device who has sent the packet as a \texttt{RemoteXBeeDevice}
- The time in which the packet was received as an Integer

add микropython_data_received_callback(callback)
Adds a callback for the event \texttt{digi.xbee.reader.MicroPythonDataReceived}.

\textbf{Parameters} \texttt{callback (Function)} – the callback. Receives one argument.

- The MicroPython data as a Bytearray

add_modem_status_received_callback(callback)
Adds a callback for the event \texttt{digi.xbee.reader.ModemStatusReceived}.

\textbf{Parameters} \texttt{callback (Function)} – the callback. Receives one argument.

- The modem status as a \texttt{digi.xbee.models.status.ModemStatus}

add_packet_received_callback(callback)
Adds a callback for the event \texttt{digi.xbee.reader.PacketReceived}.

\textbf{Parameters} \texttt{callback (Function)} – the callback. Receives two arguments.

- The received packet as a \texttt{digi.xbee.packets.base.XBeeAPIPacket}

add_route_received_callback(callback)
Adds a callback for the event \texttt{RouteReceived}. This works for Zigbee and Digimesh devices.

\textbf{Parameters} \texttt{callback (Function)} – the callback. Receives three arguments.

- source (\texttt{XBeeDevice}): The source node.
- destination (\texttt{RemoteXBeeDevice}): The destination node.
- hops (List): List of intermediate hops from closest to source to closest to destination (\texttt{RemoteXBeeDevice}).

See also:

\texttt{XBeeDevice.del_route_received_callback()}

390 Chapter 2. Contents
add_socket_data_received_callback(callback)

Adds a callback for the event `digi.xbee.reader.SocketDataReceived`.

**Parameters**

`callback` (*Function*) – the callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray

add_socket_data_received_from_callback(callback)

Adds a callback for the event `digi.xbee.reader.SocketDataReceivedFrom`.

**Parameters**

`callback` (*Function*) – the callback. Receives three arguments.

- The socket ID as an Integer.
- A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
- The data received as Bytearray

add_socket_state_received_callback(callback)

Adds a callback for the event `digi.xbee.reader.SocketStateReceived`.

**Parameters**

`callback` (*Function*) – the callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a `SocketState`

add_user_data_relay_received_callback(callback)

Adds a callback for the event `digi.xbee.reader.RelayDataReceived`.

**Parameters**

`callback` (*Function*) – the callback. Receives one argument.

- The relay data as a `digi.xbee.models.message.UserDataRelayMessage`

apply_changes()

Applies changes via AC command.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

apply_profile(profile_path, progress_callback=None)

Applies the given XBee profile to the XBee device.

**Parameters**

- `profile_path` (*String*) – path of the XBee profile file to apply.
- `progress_callback` (*Function*, optional) – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

**Raises**
• **XBeeException** – if the device is not open.
• **InvalidOperatingModeException** – if the device operating mode is invalid.
• **UpdateProfileException** – if there is any error applying the XBee profile.
• **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

**close()**
Closes the communication with the XBee device.

This method guarantees that all threads running are stopped and the serial port is closed.

**comm_iface**
Returns the hardware interface associated to the XBee device.

**Returns** the hardware interface associated to the XBee device.

**Return type** XBeeCommunicationInterface

See also:

**XBeeSerialPort**

**classmethod create_xbee_device**(comm_port_data)
Creates and returns an XBeeDevice from data of the port to which is connected.

**Parameters**
• **comm_port_data**(Dictionary) – dictionary with all comm port data needed.
• **dictionary keys are (The)**
  “baudRate” → Baud rate.
  ”port” → Port number.
  ”bitSize” → Bit size.
  ”stopBits” → Stop bits.
  ”parity” → Parity.
  ”flowControl” → Flow control.
  ”timeout” for → Timeout for synchronous operations (in seconds).

**Returns** the XBee device created.

**Return type** XBeeDevice

**Raises** **SerialException** – if the port you want to open does not exist or is already opened.

See also:

**XBeeDevice**

**del_bluetooth_data_received_callback**(callback)
Deletes a callback for the callback list of digi.xbee.reader.BluetoothDataReceived event.

**Parameters** **callback**(Function) – the callback to delete.
del_data_received_callback\(\texttt{\textbf{\{callback\}}}\)  
Deletes a callback for the callback list of `digi.xbee.reader.DataReceived` event.  

Parameters `\texttt{\textbf{\{callback\}}}` (Function) – the callback to delete.  

Raises `ValueError` – if `\texttt{\textbf{\{callback\}}}` is not in the callback list of `digi.xbee.reader.DataReceived` event.

**del_expl_data_received_callback\(\texttt{\textbf{\{callback\}}}\)**  
Deletes a callback for the callback list of `digi.xbee.reader.ExplicitDataReceived` event.  

Parameters `\texttt{\textbf{\{callback\}}}` (Function) – the callback to delete.  

Raises `ValueError` – if `\texttt{\textbf{\{callback\}}}` is not in the callback list of `digi.xbee.reader.ExplicitDataReceived` event.

**del_io_sample_received_callback\(\texttt{\textbf{\{callback\}}}\)**  
Deletes a callback for the callback list of `digi.xbee.reader.IOSampleReceived` event.  

Parameters `\texttt{\textbf{\{callback\}}}` (Function) – the callback to delete.  

Raises `ValueError` – if `\texttt{\textbf{\{callback\}}}` is not in the callback list of `digi.xbee.reader.IOSampleReceived` event.

**del_micropython_data_received_callback\(\texttt{\textbf{\{callback\}}}\)**  
Deletes a callback for the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.  

Parameters `\texttt{\textbf{\{callback\}}}` (Function) – the callback to delete.  

Raises `ValueError` – if `\texttt{\textbf{\{callback\}}}` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

**del_modem_status_received_callback\(\texttt{\textbf{\{callback\}}}\)**  
Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.  

Parameters `\texttt{\textbf{\{callback\}}}` (Function) – the callback to delete.  

Raises `ValueError` – if `\texttt{\textbf{\{callback\}}}` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

**del_packet_received_callback\(\texttt{\textbf{\{callback\}}}\)**  
Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.  

Parameters `\texttt{\textbf{\{callback\}}}` (Function) – the callback to delete.  

Raises `ValueError` – if `\texttt{\textbf{\{callback\}}}` is not in the callback list of `digi.xbee.reader.PacketReceived` event.

**del_route_received_callback\(\texttt{\textbf{\{callback\}}}\)**  
Deletes a callback for the callback list of `RouteReceived` event.  

Parameters `\texttt{\textbf{\{callback\}}}` (Function) – the callback to delete.  

See also:  

`XBeeDevice.add_route_received_callback()`
del_socket_data_received_callback(callback)
Deletes a callback for the callback list of \texttt{digi.xbee.reader.SocketDataReceived} event.

\textbf{Parameters} callback (Function) – the callback to delete.

\textbf{Raises} \texttt{ValueError} – if \texttt{callback} is not in the callback list of \texttt{digi.xbee.reader.SocketDataReceived} event.

del_socket_data_received_from_callback(callback)
Deletes a callback for the callback list of \texttt{digi.xbee.reader.SocketDataReceivedFrom} event.

\textbf{Parameters} callback (Function) – the callback to delete.

\textbf{Raises} \texttt{ValueError} – if \texttt{callback} is not in the callback list of \texttt{digi.xbee.reader.SocketDataReceivedFrom} event.

del_socket_state_received_callback(callback)
Deletes a callback for the callback list of \texttt{digi.xbee.reader.SocketStateReceived} event.

\textbf{Parameters} callback (Function) – the callback to delete.

\textbf{Raises} \texttt{ValueError} – if \texttt{callback} is not in the callback list of \texttt{digi.xbee.reader.SocketStateReceived} event.

del_user_data_relay_received_callback(callback)
Deletes a callback for the callback list of \texttt{digi.xbee.reader.RelayDataReceived} event.

\textbf{Parameters} callback (Function) – the callback to delete.

\textbf{Raises} \texttt{ValueError} – if \texttt{callback} is not in the callback list of \texttt{digi.xbee.reader.RelayDataReceived} event.

\textbf{disable_bluetooth}()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

\textbf{Raises}

\begin{itemize}
  \item \texttt{TimeoutException} – if the response is not received before the read timeout expires.
  \item \texttt{XBeeException} – if the XBee device’s serial port is closed.
  \item \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
\end{itemize}

\textbf{enable_apply_changes}(value)
Sets the apply\_changes flag.

\textbf{Parameters} value (Boolean) – True to enable the apply changes flag, False to disable it.

\textbf{enable_bluetooth}()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the \texttt{AbstractXBeeDevice.update_bluetooth_password()} method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

\textbf{Raises}

\begin{itemize}
  \item \texttt{TimeoutException} – if the response is not received before the read timeout expires.
  \item \texttt{XBeeException} – if the XBee device’s serial port is closed.
  \item \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
\end{itemize}
execute_command\((\text{parameter})\)

Executes the provided command.

Parameters \text{parameter} (\text{String}) – The name of the AT command to be executed.

Raises

- \text{TimeoutException} – if the response is not received before the read timeout expires.
- \text{XBeeException} – if the XBee device’s serial port is closed.
- \text{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \text{ATCommandException} – if the response is not as expected.

flush_queues()

Flushes the packets queue.

get_16bit_addr()

Returns the 16-bit address of the XBee device.

Returns the 16-bit address of the XBee device.

Return type XBee16BitAddress

See also:

- XBee16BitAddress

get_64bit_addr()

Returns the 64-bit address of the XBee device.

Returns the 64-bit address of the XBee device.

Return type XBee64BitAddress

See also:

- XBee64BitAddress

get_adc_value\((\text{io\_line})\)

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use \text{AbstractXBeeDevice.set\_io\_configuration()} and IOMode.ADC.

Parameters \text{io\_line} (IOLine) – the IO line to get its ADC value.

Returns the analog value corresponding to the provided IO line.

Return type Integer

Raises

- \text{TimeoutException} – if the response is not received before the read timeout expires.
- \text{XBeeException} – if the XBee device’s serial port is closed.
- \text{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ATCommandException` – if the response is not as expected.

• `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

**IOLine**

**get_api_output_mode()**

Deprecated since version 1.3: Use `get_api_output_mode_value()`

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

**Returns** the API output mode of the XBee device.

**Return type** `APIOutputMode`

**Raises**

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

See also:

**APIOutputMode**

**get_api_output_mode_value()**

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

**Returns** the parameter value.

**Return type** `Bytearray`

**Raises**

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

• `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:
get_bluetooth_mac_addr()
Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

Returns The Bluetooth MAC address.
Return type String
Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

current_frame_id()
Returns the last used frame ID.

Returns the last used frame ID.
Return type Integer

g_get_dest_address()
Returns the 64-bit address of the XBee device that data will be reported to.

Returns the address.
Return type XBee64BitAddress
Raises TimeoutException – if the response is not received before the read timeout expires.
See also:

XBee64BitAddress

g_get_dio_value(io_line)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use AbstractXBeeDevice.set_io_configuration().

Parameters io_line (IOLine) – the DIO line to gets its digital value.
Returns current value of the provided IO line.
Return type IOValue
Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the response does not contain the value for the given IO line.

See also:

IOLine
IOValue

get_firmware_version()  
Returns the firmware version of the XBee device.

**Returns** the hardware version of the XBee device.

**Return type** Bytearray

get_hardware_version()  
Returns the hardware version of the XBee device.

**Returns** the hardware version of the XBee device.

**Return type** HardwareVersion

See also:

HardwareVersion

get io_configuration(io_line)  
Returns the configuration of the provided IO line.

**Parameters** io_line (IOLine) – the io line to configure.

**Returns** the IO mode of the IO line provided.

**Return type** IOMode

**Raises**

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

• OperationNotSupportedException – if the received data is not an IO mode.

get io_sampling_rate()  
Returns the IO sampling rate of the XBee device.

**Returns** the IO sampling rate of XBee device.

**Return type** Integer

**Raises**

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.
• \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• \texttt{ATCommandException} – if the response is not as expected.

\textbf{get\_network()}

Returns this XBee device’s current network.

\begin{tabular}{l}
\textbf{Returns} \ XBeeDevice.XBeeNetwork
\end{tabular}

\textbf{get\_next\_frame\_id()}

Returns the next frame ID of the XBee device.

\begin{tabular}{l}
\textbf{Returns} \ The next frame ID of the XBee device.
\textbf{Return type} \ Integer
\end{tabular}

\textbf{get\_node\_id()}

Returns the Node Identifier (NI) value of the XBee device.

\begin{tabular}{l}
\textbf{Returns} \ the Node Identifier (NI) of the XBee device.
\textbf{Return type} \ String
\end{tabular}

\textbf{get\_pan\_id()}

Returns the operating PAN ID of the XBee device.

\begin{tabular}{l}
\textbf{Returns} \ operating PAN ID of the XBee device.
\textbf{Return type} \ Bytearray
\end{tabular}

\begin{tabular}{l}
\textbf{Raises} \ TimeoutException – if the response is not received before the read timeout expires.
\end{tabular}

\textbf{get\_parameter (param, parameter\_value=None)}

Override.

\begin{tabular}{l}
\textbf{See also:}
AbstractXBeeDevice.get\_parameter()
\end{tabular}

\textbf{get\_power\_level()}

Returns the power level of the XBee device.

\begin{tabular}{l}
\textbf{Returns} \ the power level of the XBee device.
\textbf{Return type} \ PowerLevel
\end{tabular}

\begin{tabular}{l}
\textbf{Raises} \ TimeoutException – if the response is not received before the read timeout expires.
\end{tabular}

\begin{tabular}{l}
\textbf{See also:}
PowerLevel
\end{tabular}

\textbf{get\_pwm\_duty\_cycle (io\_line)}

Returns the PWM duty cycle in % corresponding to the provided IO line.

\begin{tabular}{l}
\textbf{Parameters} \ \textbf{io\_line (IOLine)} – the IO line to get its PWM duty cycle.
\textbf{Returns} \ the PWM duty cycle of the given IO line or None if the response is empty.
\textbf{Return type} \ Integer
\end{tabular}
 Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **ValueError** – if the passed `IO_LINE` has no PWM capability.

See also:

**IOLine**

`get_role()`  
Gets the XBee role.

**Returns**  
the role of the XBee.

**Return type**  
digi.xbee.models.protocol.Role

See also:

**digi.xbee.models.protocol.Role**

`get_route_to_node(remote, timeout=10)`  
Gets the route from this XBee to the given remote node.

**Parameters**

- **remote** (`RemoteXBeeDevice`) – The remote node.
- **timeout** (`Float, optional, default=10`) – Maximum number of seconds to wait for the route.

**Returns**

Tuple containing route data (None if the route was not read in the provided timeout):
- `source` (`RemoteXBeeDevice`): The source node of the route.
- `destination` (`RemoteXBeeDevice`): The destination node of the route.
- `hops` (List): List of intermediate nodes (`RemoteXBeeDevice`) ordered from closest to source to closest to destination node (source and destination not included).

**Return type**  
Tuple

`get_sync_ops_timeout()`  
Returns the serial port read timeout.

**Returns**  
the serial port read timeout in seconds.

**Return type**  
Integer
get_xbee_device_callbacks()
Returns this XBee internal callbacks for process received packets.
This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks
will be executed before user callbacks.

    Returns PacketReceived

has_explicit_packets()
Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit
packets.

    Returns True if this XBee device’s queue has explicit packets, False otherwise.
    Return type Boolean

See also:
XBeeDevice.has_packets()

has_packets()
Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

    Returns True if this XBee device’s queue has packets, False otherwise.
    Return type Boolean

See also:
XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()
Returns whether the apply_changes flag is enabled or not.

    Returns True if the apply_changes flag is enabled, False otherwise.
    Return type Boolean

is_open()
Returns whether this XBee device is open or not.

    Returns Boolean. True if this XBee device is open, False otherwise.

is_remote()
Override method.

    See also:
AbstractXBeeDevice.is_remote()

log
Returns the XBee device log.

    Returns the XBee device logger.
    Return type Logger
operating_mode
Returns this XBee device’s operating mode.

    Returns OperatingMode. This XBee device’s operating mode.

reachable
Returns whether the XBee is reachable.

    Returns True if the device is reachable, False otherwise.
    Return type Boolean

read_data (timeout=None)
Reads new data received by this XBee device.

if a timeout is specified, this method blocks until new data is received or the timeout expires, throwing in that case a TimeoutException.

Parameters timeout (Integer, optional) – read timeout in seconds. If it’s None, this method is non-blocking and will return None if there is no data available.

    Returns the read message or None if this XBee did not receive new data.
    Return type XBeeMessage
    Raises
    • ValueError – if a timeout is specified and is less than 0.
    • TimeoutException – if a timeout is specified and no data was received during that time.
    • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
    • XBeeException – if the XBee device’s serial port is closed.

See also:

XBeeMessage

read_data_from (remote_xbee_device, timeout=None)
Reads new data received from the given remote XBee device.

If a timeout is specified, this method blocks until new data is received or the timeout expires, throwing in that case a TimeoutException.

Parameters

    • remote_xbee_device (RemoteXBeeDevice) – the remote XBee device that sent the data.
    • timeout (Integer, optional) – read timeout in seconds. If it’s None, this method is non-blocking and will return None if there is no data available.

Returns

the read message sent by remote_xbee_device or None if this XBee did not receive new data.

Return type XBeeMessage

Raises
• **ValueError** – if a timeout is specified and is less than 0.

• **TimeoutException** – if a timeout is specified and no data was received during that time.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if the XBee device’s serial port is closed.

See also:

```python
XBeeMessage
RemoteXBeeDevice
```

### read_device_info (init=True, fire_event=True)

Updates all instance parameters reading them from the XBee device.

**Parameters**

- `init (Boolean, optional, default='True')` – If False only not initialized parameters are read, all if True.

- `fire_event (Boolean, optional, default='True')` – True to launch and update event if any parameter changed, False otherwise.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.

### read_io_sample ()

Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

**Return type** `IOSample`

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.

See also:

```python
IOSample
```
reset()

Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter

Returns the scan counter for this node.

Returns  The scan counter for this node.

Return type  Integer

send_bluetooth_data(data)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters  data (Bytearray) – Data to send.

Raises

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• XBeeException – if there is any problem sending the data.

See also:

XBeeDevice.send_micropython_data()
XBeeDevice.send_user_data_relay()

send_data(remote_xbee_device, data, transmit_options=0)

Blocking method. This method sends data to a remote XBee device synchronously.

This method will wait for the packet response.

The default timeout for this method is XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS.

Parameters

• remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to send data to.

• data (String or Bytearray) – the raw data to send.

• transmit_options (Integer, optional) – transmit options, bitfield of TransmitOptions. Default to TransmitOptions.NONE.value.

Returns  XBeePacket the response.

Raises

• ValueError – if remote_xbee_device is None.

• TimeoutException – if this method can’t read a response packet in XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS seconds.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• *TransmitException* – if the status of the response received is not OK.

• *XBeeException* – if the XBee device’s serial port is closed.

See also:

RemoteXBeeDevice
XBeePacket

**send_data_async** (*remote_xbee_device, data, transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee device.

This method won’t wait for the response.

**Parameters**

• **remote_xbee_device** (*RemoteXBeeDevice*) – the remote XBee device to send data to.

• **data** (*String or Bytearray*) – the raw data to send.

• **transmit_options** (*Integer, optional*) – transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

**Raises**

• *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• *XBeeException* – if the XBee device’s serial port is closed.

See also:

RemoteXBeeDevice

**send_data_broadcast** (*data, transmit_options=0*)

Sends the provided data to all the XBee nodes of the network (broadcast).

This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using the *AbstractXBeeDevice.set_sync_ops_timeout()* method and can be consulted with *AbstractXBeeDevice.get_sync_ops_timeout()* method.

**Parameters**

• **data** (*String or Bytearray*) – data to send.

• **transmit_options** (*Integer, optional*) – transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

**Raises**

• *TimeoutException* – if this method can’t read a response packet in *XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS* seconds.

• *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• *TransmitException* – if the status of the response received is not OK.
• **XBeeException** – if the XBee device’s serial port is closed.

**send_micropython_data** *(data)*

Sends the given data to the MicroPython interface using a User Data Relay frame.

**Parameters** *data* *(Bytearray)* – Data to send.

**Raises**

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if there is any problem sending the data.

**See also:**

*XBeeDevice.send_bluetooth_data()*

*XBeeDevice.send_user_data_relay()*

**send_packet** *(packet, sync=False)*

Override method.

**See also:**

*AbstractXBeeDevice._send_packet()*

**send_packet_sync_and_get_response** *(packet_to_send, timeout=None)*

Override method.

**See also:**

*AbstractXBeeDevice._send_packet_sync_and_get_response()*

**send_user_data_relay** *(local_interface, data)*

Sends the given data to the given XBee local interface.

**Parameters**

• *local_interface* *(XBeeLocalInterface)* – Destination XBee local interface.

• *data* *(Bytearray)* – Data to send.

**Raises**

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ValueError** – if *local_interface* is None.

• **XBeeException** – if there is any problem sending the User Data Relay.

**See also:**

*XBeeLocalInterface*
serial_port

Returns the serial port associated to the XBee device, if any.

Returns

the serial port associated to the XBee device. Returns None if the local XBee does not use serial communication.

Return type XBeeSerialPort

See also:

XBeeSerialPort

set_16bit_addr(value)

Sets the 16-bit address of the XBee device.

Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is not 802.15.4.

set_api_output_mode(api_output_mode)

Deprecated since version 1.3: Use set_api_output_mode_value() Sets the API output mode of the XBee device.

Parameters api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is ZigBee

See also:

APIOutputMode

set_api_output_mode_value(api_output_mode)

Sets the API output mode of the XBee.
Parameters `api_output_mode (Integer)` – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

`set_dest_address (addr)`
Sets the 64-bit address of the XBee device that data will be reported to.

**Parameters**

- `addr (XBee64BitAddress or RemoteXBeeDevice)` – the address itself or the remote XBee device that you want to set up its address as destination address.

**Raises**

- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If the response is not as expected.
- `ValueError` – If `addr` is None.

`set_dio_change_detection (io_lines_set)`
Sets the digital IO lines to be monitored and sampled whenever their status changes.

A `None` set of lines disables this feature.

**Parameters**

- `io_lines_set` – set of `IOLine`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:
**IOLine**

**set_dio_value**(io_line, io_value)
Sets the digital value (high or low) to the provided IO line.

**Parameters**

- `io_line` (**IOLine**) – the digital IO line to set its value.
- `io_value` (**IOValue**) – the IO value to set to the IO line.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

**IOLine**
**IOValue**

**set_io_configuration**(io_line, io_mode)
Sets the configuration of the provided IO line.

**Parameters**

- `io_line` (**IOLine**) – the IO line to configure.
- `io_mode` (**IOMode**) – the IO mode to set to the IO line.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

**IOLine**
**IOMode**

**set_io_sampling_rate**(rate)
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

**Parameters**

- `rate` (**Integer**) – the new IO sampling rate of the XBee device in seconds.

**Raises**
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBe device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**set_node_id**(*node_id*)

Sets the Node Identifier (NI) value of the XBe device.

**Parameters**
- **node_id** *(String)* – the new Node Identifier (NI) of the XBe device.

**Raises**
- **ValueError** – if `node_id` is `None` or its length is greater than 20.
- **TimeoutException** – if the response is not received before the read timeout expires.

**set_pan_id**(*value*)

Sets the operating PAN ID of the XBe device.

**Parameters**
- **value** *(Bytearray)* – the new operating PAN ID of the XBe device. Must have only 1 or 2 bytes.

**Raises**
- **TimeoutException** – if the response is not received before the read timeout expires.

**set_power_level**(*power_level*)

Sets the power level of the XBe device.

**Parameters**
- **power_level** *(PowerLevel)* – the new power level of the XBe device.

**Raises**
- **TimeoutException** – if the response is not received before the read timeout expires.

**set_parameter**(*param, value*)

Override.

See: AbstractXBeeDevice.set.parameter()

**set_pwm_duty_cycle**(*io_line, cycle*)

Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

**Parameters**
- **io_line** *(IOLine)* – the IO Line to be assigned.
- **cycle** *(Integer)* – duty cycle in % to be assigned. Must be between 0 and 100.

**Raises**
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBe device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
• **ValueError** – if the given IO line does not have PWM capability or `cycle` is not between 0 and 100.

See also:

```python
IOLine
IOMode.PWM
```

**set_sync_ops_timeout**(*sync_ops_timeout*)

Sets the serial port read timeout.

**Parameters**

- **sync_ops_timeout** *(Integer)* – the read timeout, expressed in seconds.

**update_bluetooth_password**(*new_password*)

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**

- **new_password** *(String)* – New Bluetooth password.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from**(*device*)

Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**

- **device** *(AbstractXBeeDevice)* – the XBee device to get the data from.

**Returns**

- **True** if the device data has been updated, **False** otherwise.

**Return type**

- **Boolean**

**update_firmware**(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the device.

**Parameters**

- **xml_firmware_file** *(String)* – path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** *(String, optional)* – location of the XBee binary firmware file.
- **bootloader_firmware_file** *(String, optional)* – location of the bootloader binary firmware file.
- **timeout** *(Integer, optional)* – the maximum time to wait for target read operations during the update process.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
-- The current update task percentage as an Integer

 Raises

- XBeeException – if the device is not open.
- InvalidOperatingModeException – if the device operating mode is invalid.
- OperationNotSupportedException – if the firmware update is not supported in the XBee device.
- FirmwareUpdateException – if there is any error performing the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method is_apply_configuration_changes_enabled() to get its status and enable_apply_configuration_changes() to enable/disable the option. If it is disabled, method apply_changes() can be used in order to manually apply the changes.

 Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

class digi.xbee.devices.DigiPointDevice (port=None, baud_rate=None, data_bits=sphinx.ext.autodoc.importer._MockObject object>, stop_bits=sphinx.ext.autodoc.importer._MockObject object>, parity=sphinx.ext.autodoc.importer._MockObject object>, flow_control=FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)

Bases: digi.xbee.devices.XBeeDevice

This class represents a local DigiPoint XBee device.

Class constructor. Instantiates a new DigiPointDevice with the provided parameters.

 Parameters

- **port** (Integer or String) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.
- **baud_rate** (Integer) – the serial port baud rate.
- **data_bits** (Integer, default (_sync_ops_timeout) – serial.EIGHTBITS): comm port bitsize.
- **stop_bits** (Integer, default – serial.STOPBITS_ONE): comm port stop bits.
• (Character, default (parity) – serial.PARITY_NONE): comm port parity.
• (Integer, default – FlowControl.NONE): comm port flow control.
• (Integer, default – 3): the read timeout (in seconds).
• comm_iface (XBeeCommunicationInterface) – the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor:

See also:

XBeeDevice
XBeeDevice.__init__()

open (force_settings=False)
Override.

Raises

• TimeoutException – If there is any problem with the communication.
• InvalidOperatingModeException – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• XBeeException – If the protocol is invalid or if the XBee device is already open.

See also:

XBeeDevice.open()

get_protocol()
Override.

See also:

XBeeDevice.get_protocol()

send_data_64_16 (x64addr, x16addr, data, transmit_options=0)
Override.

See also:

XBeeDevice.send_data_64_16()

send_data_async_64_16 (x64addr, x16addr, data, transmit_options=0)
Override.

See also:

XBeeDevice.send_data_async_64_16()
read_expl_data(timeout=None)
    Override.
    
    See also:

    XBeeDevice.read_expl_data()

read_expl_data_from(remote_xbee_device, timeout=None)
    Override.
    
    See also:

    XBeeDevice.read_expl_data_from()

send_expl_data(remote_xbee_device, data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)
    Override.
    
    See also:

    XBeeDevice.send_expl_data()

send_expl_data_broadcast(data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)
    Override.
    
    See also:

    XBeeDevice._send_expl_data_broadcast()

send_expl_data_async(remote_xbee_device, data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)
    Override.
    
    See also:

    XBeeDevice.send_expl_data_async()

add_bluetooth_data_received_callback(callback)
    Adds a callback for the event digi.xbee.reader.BluetoothDataReceived.

    Parameters callback (Function) – the callback. Receives one argument.
    - The Bluetooth data as a Bytarray

add_data_received_callback(callback)
    Adds a callback for the event digi.xbee.reader.DataReceived.

    Parameters callback (Function) – the callback. Receives one argument.
• The data received as an `digi.xbee.models.message.XBeeMessage`

`add_expl_data_received_callback(callback)`
Add a callback for the event `digi.xbee.reader.ExplicitDataReceived`.

**Parameters**

`callback (Function)` – the callback. Receives one argument.

• The explicit data received as a `digi.xbee.models.message.ExplicitXBeeMessage`.

`add_io_sample_received_callback(callback)`
Add a callback for the event `digi.xbee.reader.IOSampleReceived`.

**Parameters**

`callback (Function)` – the callback. Receives three arguments.

• The received IO sample as an `digi.xbee.io.IOSample`
• The remote XBee device who has sent the packet as a `RemoteXBeeDevice`
• The time in which the packet was received as an Integer

`add_micropython_data_received_callback(callback)`
Add a callback for the event `digi.xbee.reader.MicroPythonDataReceived`.

**Parameters**

`callback (Function)` – the callback. Receives one argument.

• The MicroPython data as a Bytearray

`add_modem_status_received_callback(callback)`
Add a callback for the event `digi.xbee.reader.ModemStatusReceived`.

**Parameters**

`callback (Function)` – the callback. Receives one argument.

• The modem status as a `digi.xbee.models.status.ModemStatus`

`add_packet_received_callback(callback)`
Add a callback for the event `digi.xbee.reader.PacketReceived`.

**Parameters**

`callback (Function)` – the callback. Receives two arguments.

• The received packet as a `digi.xbee.packets.base.XBeeAPIPacket`

`add_route_received_callback(callback)`
Add a callback for the event `RouteReceived`. This works for Zigbee and Digimesh devices.

**Parameters**

`callback (Function)` – the callback. Receives three arguments.

• source (`XBeeDevice`): The source node.
• destination (`RemoteXBeeDevice`): The destination node.
• hops (List): List of intermediate hops from closest to source to closest to destination (`RemoteXBeeDevice`).

See also:

`XBeeDevice.del_route_received_callback()`

`add_socket_data_received_callback(callback)`
Add a callback for the event `digi.xbee.reader.SocketDataReceived`.

**Parameters**

`callback (Function)` – the callback. Receives two arguments.

• The socket ID as an Integer.
• The data received as Bytarray

**add_socket_data_received_from_callback**(*callback*)
 Adds a callback for the event `digi.xbee.reader.SocketDataReceivedFrom`.

**Parameters**
*callback* (*Function*) – the callback. Receives three arguments.

  • The socket ID as an Integer.
  
  • A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
  
  • The data received as Bytarray

**add_socket_state_received_callback**(*callback*)
 Adds a callback for the event `digi.xbee.reader.SocketStateReceived`.

**Parameters**
*callback* (*Function*) – the callback. Receives two arguments.

  • The socket ID as an Integer.
  
  • The state received as a `SocketState`

**add_user_data_relay_received_callback**(*callback*)
 Adds a callback for the event `digi.xbee.reader.RelayDataReceived`.

**Parameters**
*callback* (*Function*) – the callback. Receives one argument.

  • The relay data as a `digi.xbee.models.message.UserDataRelayMessage`

**apply_changes**()
 Applies changes via AC command.

**Raises**

  • `TimeoutException` – if the response is not received before the read timeout expires.
  
  • `XBeeException` – if the XBe device’s serial port is closed.
  
  • `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  
  • `ATCommandException` – if the response is not as expected.

**apply_profile**(*profile_path*, *progress_callback=None*)
 Applies the given XBe profile to the XBe device.

**Parameters**

  • *profile_path* (*String*) – path of the XBe profile file to apply.
  
  • *progress_callback* (*Function, optional*) – function to execute to receive progress information. Receives two arguments:

    – The current apply profile task as a String
    – The current apply profile task percentage as an Integer

**Raises**

  • `XBeeException` – if the device is not open.
  
  • `InvalidOperatingModeException` – if the device operating mode is invalid.
  
  • `UpdateProfileException` – if there is any error applying the XBe profile.
• **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

close()
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

comm_iface
Returns the hardware interface associated to the XBee device.

Returns the hardware interface associated to the XBee device.

Return type **XBeeCommunicationInterface**

See also:

**XBeeSerialPort**

**classmethod create_xbee_device**(comm_port_data)
Creates and returns an **XBeeDevice** from data of the port to which is connected.

Parameters

- **comm_port_data**(Dictionary) – dictionary with all comm port data needed.

  • **dictionary keys are** (The) –
    - "baudRate" –> Baud rate.
    - "port" –> Port number.
    - "bitSize" –> Bit size.
    - "stopBits" –> Stop bits.
    - "parity" –> Parity.
    - "flowControl" –> Flow control.
    - "timeout" for –> Timeout for synchronous operations (in seconds).

Returns the XBee device created.

Return type **XBeeDevice**

Raises **SerialException** – if the port you want to open does not exist or is already opened.

See also:

**XBeeDevice**

del_bluetooth_data_received_callback**(callback)
Deletes a callback for the callback list of **digi.xbee.readerBluetoothDataReceived** event.

Parameters **callback**(Function) – the callback to delete.

Raises **ValueError** – if **callback** is not in the callback list of **digi.xbee.readerBluetoothDataReceived** event.

del_data_received_callback**(callback)
Deletes a callback for the callback list of **digi.xbee.readerDataReceived** event.

Parameters **callback**(Function) – the callback to delete.
Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.DataReceived` event.

`del_expl_data_received_callback` *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.ExplicitDataReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ExplicitDataReceived` event.

`del_io_sample_received_callback` *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.IOSampleReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.IOSampleReceived` event.

`del_micropython_data_received_callback` *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

`del_modem_status_received_callback` *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

`del_packet_received_callback` *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.PacketReceived` event.

`del_route_received_callback` *(callback)*
Deletes a callback for the callback list of `RouteReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

See also:

`XBeeDevice.add_route_received_callback()`

`del_socket_data_received_callback` *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.
del_socket_data_received_from_callback (callback)
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

del_socket_state_received_callback (callback)
Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

del_user_data_relay_received_callback (callback)
Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.

disable_bluetooth ()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

enable_apply_changes (value)
Sets the apply_changes flag.

Parameters value (Boolean) – True to enable the apply changes flag, False to disable it.

enable_bluetooth ()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

eexecute_command (parameter)
Executes the provided command.

Parameters parameter (String) – The name of the AT command to be executed.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

flush_queues()
    Flushes the packets queue.

get_16bit_addr()
    Returns the 16-bit address of the XBee device.
    Returns the 16-bit address of the XBee device.
    Return type XBee16BitAddress
    See also:

    XBee16BitAddress

get_64bit_addr()
    Returns the 64-bit address of the XBee device.
    Returns the 64-bit address of the XBee device.
    Return type XBee64BitAddress
    See also:

    XBee64BitAddress

get_adc_value(io_line)
    Returns the analog value of the provided IO line.
    The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.set_io_configuration() and IOMode.ADC.
    Parameters io_line (IOLine) – the IO line to get its ADC value.
    Returns the analog value corresponding to the provided IO line.
    Return type Integer
    Raises
    • TimeoutException – if the response is not received before the read timeout expires.
    • XBeeException – if the XBee device’s serial port is closed.
    • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
    • ATCommandException – if the response is not as expected.
    • OperationNotSupportedException – if the response does not contain the value for the given IO line.
    See also:
get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

Returns  the API output mode of the XBee device.

Return type  APIOutputMode

Raises

-  `TimeoutException` – if the response is not received before the read timeout expires.
-  `XBeeException` – if the XBee device’s serial port is closed.
-  `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
-  `ATCommandException` – if the response is not as expected.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns  the parameter value.

Return type  Bytearray

Raises

-  `TimeoutException` – if the response is not received before the read timeout expires.
-  `XBeeException` – if the XBee device’s serial port is closed.
-  `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
-  `ATCommandException` – if the response is not as expected.
-  `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:
**get_bluetooth_mac_addr()**

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

**Returns**  The Bluetooth MAC address.

**Return type**  String

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the Xbee device’s serial port is closed.
- `InvalidOperatingModeException` – if the Xbee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**get_current_frame_id()**

Returns the last used frame ID.

**Returns**  the last used frame ID.

**Return type**  Integer

**get_dest_address()**

Returns the 64-bit address of the XBee device that data will be reported to.

**Returns**  the address.

**Return type**  XBee64BitAddress

**Raises**  `TimeoutException` – if the response is not received before the read timeout expires.

**See also:**  `XBee64BitAddress`

**get_dio_value(io_line)**

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

**Parameters**  `io_line` (IOLine) – the DIO line to gets its digital value.

**Returns**  current value of the provided IO line.

**Return type**  IOVvalue

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the Xbee device’s serial port is closed.
- `InvalidOperatingModeException` – if the Xbee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.
See also:

IOLine
IOValue

get_firmware_version() Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type Bytearray

get_hardware_version() Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type HardwareVersion

See also:

HardwareVersion

get_io_configuration(io_line) Returns the configuration of the provided IO line.

Parameters io_line (IOLine) – the io line to configure.

Returns the IO mode of the IO line provided.

Return type IOMode

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBe device’s serial port is closed.
• InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the received data is not an IO mode.

get_io_sampling_rate() Returns the IO sampling rate of the XBee device.

Returns the IO sampling rate of XBee device.

Return type Integer

 Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBe device’s serial port is closed.
• InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

**get_network()**

Returns this XBee device’s current network.

Returns **XBeeDevice.XBeeNetwork**

**get_next_frame_id()**

Returns the next frame ID of the XBee device.

Returns The next frame ID of the XBee device.

Return type Integer

**get_node_id()**

Returns the Node Identifier (NI) value of the XBee device.

Returns the Node Identifier (NI) of the XBee device.

Return type String

**get_pan_id()**

Returns the operating PAN ID of the XBee device.

Returns operating PAN ID of the XBee device.

Return type Bytearray

Raises **TimeoutException** – if the response is not received before the read timeout expires.

**get_parameter** *(param, parameter_value=None)*

Override.

See also:

```
AbstractXBeeDevice.get_parameter()
```

**get_power_level()**

Returns the power level of the XBee device.

Returns the power level of the XBee device.

Return type **PowerLevel**

Raises **TimeoutException** – if the response is not received before the read timeout expires.

See also:

```
PowerLevel
```

**get_pwm_duty_cycle** *(io_line)*

Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters **io_line** *(IOLine)* – the IO line to get its PWM duty cycle.

Returns the PWM duty cycle of the given IO line or **None** if the response is empty.

Return type Integer

Raises
• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **ValueError** – if the passed IO_LINE has no PWM capability.

See also:

IOLine

get_role()  
Gets the XBee role.

**Returns**  
the role of the XBee.

**Return type**  
digi.xbee.models.protocol.Role

See also:

digi.xbee.models.protocol.Role

get_route_to_node(remote, timeout=10)  
Gets the route from this XBee to the given remote node.

**Parameters**

• **remote (RemoteXBeeDevice)** – The remote node.

• **timeout (Float, optional, default=10)** – Maximum number of seconds to wait for the route.

**Returns**

Tuple containing route data (None if the route was not read in the provided timeout):

- **source (RemoteXBeeDevice)**: The source node of the route.
- **destination (RemoteXBeeDevice)**: The destination node of the route.

• hops (List): List of intermediate nodes (RemoteXBeeDevice) ordered from closest to source to closest to destination node (source and destination not included).

**Return type**  
Tuple

get_sync_ops_timeout()  
Returns the serial port read timeout.

**Returns**  
the serial port read timeout in seconds.

**Return type**  
Integer
get_xbee_device_callbacks()  
Returns this XBee internal callbacks for process received packets.
This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

Returns PacketReceived

has_explicit_packets()  
Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

Returns True if this XBee device’s queue has explicit packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()  
Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

Returns True if this XBee device’s queue has packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()  
Returns whether the apply_changes flag is enabled or not.

Returns True if the apply_changes flag is enabled, False otherwise.

Return type Boolean

is_open()  
Returns whether this XBee device is open or not.

Returns Boolean. True if this XBee device is open, False otherwise.

is_remote()  
Override method.

See also:

AbstractXBeeDevice.is_remote()

log  
Returns the XBee device log.

Returns the XBee device logger.

Return type Logger
operating_mode
Returns this XBee device’s operating mode.

Returns OperatingMode. This XBee device’s operating mode.

reachable
Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

read_data (timeout=None)
Reads new data received by this XBee device.

If a timeout is specified, this method blocks until new data is received or the timeout expires, throwing in that case a TimeoutException.

Parameters timeout (Integer, optional) -- read timeout in seconds. If it’s None, this method is non-blocking and will return None if there is no data available.

Returns the read message or None if this XBee did not receive new data.

Return type XBeeMessage

Raises

• ValueError – if a timeout is specified and is less than 0.

• TimeoutException – if a timeout is specified and no data was received during that time.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• XBeeException – if the XBee device’s serial port is closed.

See also:

XBeeMessage

read_data_from (remote_xbee_device, timeout=None)
Reads new data received from the given remote XBee device.

If a timeout is specified, this method blocks until new data is received or the timeout expires, throwing in that case a TimeoutException.

Parameters

• remote_xbee_device (RemoteXBeeDevice) -- the remote XBee device that sent the data.

• timeout (Integer, optional) -- read timeout in seconds. If it’s None, this method is non-blocking and will return None if there is no data available.

Returns

the read message sent by remote_xbee_device or None if this XBee did not receive new data.

Return type XBeeMessage

 Raises

2.5. API reference 427
• **ValueError** – if a timeout is specified and is less than 0.
• **TimeoutException** – if a timeout is specified and no data was received during that time.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **XBeeException** – if the XBee device’s serial port is closed.

See also:

*XBeeMessage*
*RemoteXBeeDevice*

**read_device_info**(init=True, fire_event=True)
Updates all instance parameters reading them from the XBee device.

**Parameters**

- **init** *(Boolean, optional, default='True')* – If False only not initialized parameters are read, all if True.
- **fire_event** *(Boolean, optional, default='True')* – True to launch and update event if any parameter changed, False otherwise.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**read_io_sample**()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

**Returns** the IO sample read from the XBee device.

**Return type** **IOSample**

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

*IOSample*
reset()
Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.
Return type Integer

send_bluetooth_data(data)
Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters data (Bytearray) – Data to send.

Raises
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• XBeeException – if there is any problem sending the data.

See also:
XBeeDevice.send_micropython_data()
XBeeDevice.send_user_data_relay()

send_data(remote_xbee_device, data, transmit_options=0)
Blocking method. This method sends data to a remote XBee device synchronously.

This method will wait for the packet response.

The default timeout for this method is XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS.

Parameters
• remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to send data to.
• data (String or Bytestream) – the raw data to send.
• transmit_options (Integer, optional) – transmit options, bitfield of TransmitOptions. Default to TransmitOptions.NONE.value.

Returns XBeePacket the response.

Raises
• ValueError – if remote_xbee_device is None.
• TimeoutException – if this method can’t read a response packet in XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS seconds.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• *TransmitException* – if the status of the response received is not OK.
• *XBeeException* – if the XBee device’s serial port is closed.

See also:

`RemoteXBeeDevice`  
`XBeePacket`

**send_data_async** (*remote_xbee_device, data, transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee device.

This method won’t wait for the response.

**Parameters**

• **remote_xbee_device** (*RemoteXBeeDevice*) – the remote XBee device to send data to.
• **data** (*String or Bytearray*) – the raw data to send.
• **transmit_options** (*Integer, optional*) – transmit options, bitfield of `TransmitOptions`. Default to `TransmitOptions.NONE.value`.

**Raises**

• *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• *XBeeException* – if the XBee device’s serial port is closed.

See also:

`RemoteXBeeDevice`

**send_data_broadcast** (*data, transmit_options=0*)

Sends the provided data to all the XBee nodes of the network (broadcast).

This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using the `AbstractXBeeDevice.set_sync_ops_timeout()` method and can be consulted with `AbstractXBeeDevice.get_sync_ops_timeout()` method.

**Parameters**

• **data** (*String or Bytearray*) – data to send.
• **transmit_options** (*Integer, optional*) – transmit options, bitfield of `TransmitOptions`. Default to `TransmitOptions.NONE.value`.

**Raises**

• *TimeoutException* – if this method can’t read a response packet in `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS` seconds.
• *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• *TransmitException* – if the status of the response received is not OK.
• `XBeeException` – if the XBee device’s serial port is closed.

**send_micropython_data** (*data*)

Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `XBeeException` – if there is any problem sending the data.

See also:

`XBeeDevice.send_bluetooth_data()`  
`XBeeDevice.send_user_data_relay()`

**send_packet** (*packet*, **sync**=`False`)

Override method.

See also:

AbstractXBeeDevice._send_packet()

**send_packet_sync_and_get_response** (*packet_to_send*, **timeout**=`None`)

Override method.

See also:

AbstractXBeeDevice._send_packet_sync_and_get_response()

**send_user_data_relay** (*local_interface*, **data**)

Sends the given data to the given XBee local interface.

Parameters

• **local_interface** (*XBeeLocalInterface*) – Destination XBee local interface.

• **data** (*Bytearray*) – Data to send.

Raises

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ValueError` – if `local_interface` is `None`.

• `XBeeException` – if there is any problem sending the User Data Relay.

See also:

`XBeeLocalInterface`
serial_port
Returns the serial port associated to the XBee device, if any.

Returns
the serial port associated to the XBee device. Returns None if the local XBee does not use serial communication.

Return type XBeeSerialPort
See also:
XBeeSerialPort

set_16bit_addr(value)
Sets the 16-bit address of the XBee device.

Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is not 802.15.4.

set_api_output_mode(api_output_mode)
Deprecated since version 1.3: Use set_api_output_mode_value()
Sets the API output mode of the XBee device.

Parameters api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is ZigBee

See also:
APIOutputMode

set_api_output_mode_value(api_output_mode)
Sets the API output mode of the XBee.
Parameters **api_output_mode** (*Integer*) – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **OperationNotSupportedException** – if it is not supported by the current protocol.

See also:

```python
digi.xbee.models.mode.APIOutputModeBit
```

**set_dest_address**(*addr*)

Sets the 64-bit address of the XBee device that data will be reported to.

Parameters **addr** (*XBee64BitAddress* or *RemoteXBeeDevice*) – the address itself or the remote XBee device that you want to set up its address as destination address.

Raises

- **TimeoutException** – If the response is not received before the read timeout expires.
- **XBeeException** – If the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – If the response is not as expected.
- **ValueError** – If `addr` is None.

**set_dio_change_detection**(*io_lines_set*)

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A `None` set of lines disables this feature.

Parameters **io_lines_set** – set of `IOLine`.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:
IOLine

set_dio_value (io_line, io_value)
Sets the digital value (high or low) to the provided IO line.

Parameters

- **io_line** (IOLine) – the digital IO line to sets its value.
- **io_value** (IOValue) – the IO value to set to the IO line.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBees device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

IOLine
IOValue

set_io_configuration (io_line, io_mode)
Sets the configuration of the provided IO line.

Parameters

- **io_line** (IOLine) – the IO line to configure.
- **io_mode** (IOMode) – the IO mode to set to the IO line.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBees device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

IOLine
IOMode

set_io_sampling_rate (rate)
Sets the IO sampling rate of the XBees device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (Integer) – the new IO sampling rate of the XBees device in seconds.

Raises
• `TimeoutException` – if the response is not received before the read timeout expires.
• `XBeeException` – if the XBees device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ATCommandException` – if the response is not as expected.

`set_node_id(node_id)`
Sets the Node Identifier (NI) value of the XBees device.

Parameters:
- `node_id` (*String*) – the new Node Identifier (NI) of the XBees device.

Raises:
- `ValueError` – if `node_id` is `None` or its length is greater than 20.
- `TimeoutException` – if the response is not received before the read timeout expires.

`set_pan_id(value)`
Sets the operating PAN ID of the XBees device.

Parameters:
- `value` (*Bytearray*) – the new operating PAN ID of the XBees device. Must have only 1 or 2 bytes.

Raises:
- `TimeoutException` – if the response is not received before the read timeout expires.

`set_parameter(param, value)`
Override.

See: `AbstractXBeeDevice.set_parameter()`

`set_power_level(power_level)`
Sets the power level of the XBees device.

Parameters:
- `power_level` (*PowerLevel*) – the new power level of the XBees device.

Raises:
- `TimeoutException` – if the response is not received before the read timeout expires.

See also: `PowerLevel`

`set_pwm_duty_cycle(io_line, cycle)`
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters:
- `io_line` (*IOLine*) – the IO Line to be assigned.
- `cycle` (*Integer*) – duty cycle in % to be assigned. Must be between 0 and 100.

Raises:
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
ValueError – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

IOLine
IOMode.PWM

set_sync_ops_timeout(sync_ops_timeout)
Sets the serial port read timeout.

Parameters sync_ops_timeout (Integer) – the read timeout, expressed in seconds.

update_bluetooth_password(new_password)
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

Parameters new_password (String) – New Bluetooth password.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

update_device_data_from(device)
Updates the current device reference with the data provided for the given device.

This is only for internal use.

Parameters device (AbstractXBeeDevice) – the XBee device to get the data from.

Returns True if the device data has been updated, False otherwise.

Return type Boolean

update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)
Performs a firmware update operation of the device.

Parameters

• xml_firmware_file (String) – path of the XML file that describes the firmware to upload.
• xbee_firmware_file (String, optional) – location of the XBee binary firmware file.
• bootloader_firmware_file (String, optional) – location of the bootloader binary firmware file.
• timeout (Integer, optional) – the maximum time to wait for target read operations during the update process.
• progress_callback (Function, optional) – function to execute to receive progress information. Receives two arguments:
  – The current update task as a String
– The current update task percentage as an Integer

**Raises**

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.
- **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes()**

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method **is_apply_configuration_changes_enabled()** to get its status and **enable_apply_configuration_changes()** to enable/disable the option. If it is disabled, method **apply_changes()** can be used in order to manually apply the changes.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**class** **digi.xbee.devices.ZigBeeDevice**(port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)

**Bases:** **digi.xbee.devices.XBeeDevice**

This class represents a local ZigBee XBee device.

**Class constructor.** Instantiates a new **ZigBeeDevice** with the provided parameters.

**Parameters**

- **port** (**Integer or String**) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.
- **baud_rate** (**Integer**) – the serial port baud rate.
- **data_bits** (**Integer**, default (**flow_control**) – **serial.EIGHTBITS**): comm port bit-size.
- **stop_bits** (**Integer**, default (**serial.STOPBITS_ONE**): comm port stop bits.
- **parity** (**Character**, default (**serial.PARITY_NONE**): comm port parity.

2.5. API reference
• (Integer, default = FlowControl.NONE): comm port flow control.

_sync_ops_timeout (Integer, default: 3): the read timeout (in seconds). comm_iface
(XBeeCommunicationInterface): the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor.: 

See also:

XBeeDevice
XBeeDevice.__init__() 

open (force_settings=False)
Override.

Raises

• TimeoutException – If there is any problem with the communication.

• InvalidOperatingModeException – If the XBee device’s operating mode is not
API or ESCAPED API. This method only checks the cached value of the operating mode.

• XBeeException – If the protocol is invalid or if the XBee device is already open.

See also:

XBeeDevice.open ()

get_protocol ()
Override.

See also:

XBeeDevice.get_protocol ()

get_ai_status ()
Override.

See also:

AbstractXBeeDevice._get_ai_status ()

force_disassociate ()
Override.

See also:

AbstractXBeeDevice._force_disassociate ()
get\_many\_to\_one\_broadcasting\_time()    
Returns the time between aggregation route broadcast in seconds.

**Returns**

The number of seconds between aggregation route broadcasts. -1 if it is disabled.

**Return type** Integer

**Raises**

- **TimeoutException** – If the response is not received before the read timeout expires.
- **XBeeException** – If the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – If the response is not as expected.

set\_many\_to\_one\_broadcasting\_time\_time\(\text{seconds}\)    
Configures the time between aggregation route broadcast in seconds.

**Parameters** seconds (Integer) – The number of seconds between aggregation route broadcasts. -1 to disable. 0 to only send one broadcast.

**Raises**

- **TimeoutException** – If the response is not received before the read timeout expires.
- **XBeeException** – If the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – If the response is not as expected.
- **ValueError** – If seconds is None or is lower than -1, or bigger than 254.

send\_data\_64\_16\(\text{x64addr}, \text{x16addr}, \text{data}, \text{transmit\_options=0}\)    
Override.

See also:

XBeeDevice.send\_data\_64\_16()

send\_data\_async\_64\_16\(\text{x64addr}, \text{x16addr}, \text{data}, \text{transmit\_options=0}\)    
Override.

See also:

XBeeDevice.send\_data\_async\_64\_16()

read\_expl\_data\(\text{timeout=\text{None}}\)    
Override.

See also:

XBeeDevice._read\_expl\_data()
**read_expl_data_from** *(remote_xbee_device, timeout=None)*  
Override.  
See also:  

XBeeDevice._read_expl_data_from()

**send_expl_data** *(remote_xbee_device, data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)*  
Override.  
See also:  

XBeeDevice._send_expl_data()

**send_expl_data_broadcast** *(data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)*  
Override.  
See also:  

XBeeDevice._send_expl_data_broadcast()

**send_expl_data_async** *(remote_xbee_device, data, src_endpoint, dest_endpoint, cluster_id, profile_id, transmit_options=0)*  
Override.  
See also:  

XBeeDevice.send_expl_data_async()

**send_multicast_data** *(group_id, data, src_endpoint, dest_endpoint, cluster_id, profile_id)*  
Blocking method. This method sends multicast data to the provided group ID synchronously.  
This method will wait for the packet response.  
The default timeout for this method is XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS.  

Parameters

- **group_id** *(XBee16BitAddress)* – the 16 bit address of the multicast group.
- **data** *(Bytearray)* – the raw data to send.
- **src_endpoint** *(Integer)* – source endpoint of the transmission. 1 byte.
- **dest_endpoint** *(Integer)* – destination endpoint of the transmission. 1 byte.
- **cluster_id** *(Integer)* – Cluster ID of the transmission. Must be between 0x0 and 0xFFFF.
- **profile_id** *(Integer)* – Profile ID of the transmission. Must be between 0x0 and 0xFFFF.
Returns  the response packet.

Return type  XBeePacket

Raises

- `TimeoutException` – if this method can’t read a response packet in `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS` seconds.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – if the XBee device’s serial port is closed.
- `XBeeException` – if the status of the response received is not OK.

See also:

```
XBee16BitAddress
XBeePacket
```

### send_multicast_data_async(group_id, data, src_endpoint, dest_endpoint, cluster_id, profile_id)

Non-blocking method. This method sends multicast data to the provided group ID.

This method won’t wait for the response.

Parameters

- `group_id (XBee16BitAddress)` – the 16 bit address of the multicast group.
- `data (Bytearray)` – the raw data to send.
- `src_endpoint (Integer)` – source endpoint of the transmission. 1 byte.
- `dest_endpoint (Integer)` – destination endpoint of the transmission. 1 byte.
- `cluster_id (Integer)` – Cluster ID of the transmission. Must be between 0x0 and 0xFFFF.
- `profile_id (Integer)` – Profile ID of the transmission. Must be between 0x0 and 0xFFFF.

Raises

- `TimeoutException` – if this method can’t read a response packet in `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS` seconds.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – if the XBee device’s serial port is closed.

See also:

```
XBee16BitAddress
```
register_joining_device (registrant_address, options, key)
Securely registers a joining device to a trust center. Registration is the process by which a node is authorized to join the network using a preconfigured link key or installation code that is conveyed to the trust center out-of-band (using a physical interface and not over-the-air).

This method is synchronous, it sends the register joining device packet and waits for the answer of the operation. Then, returns the corresponding status.

Parameters

- **registrant_address** (XBee64BitAddress) – the 64-bit address of the device to register.
- **options** (RegisterKeyOptions) – the register options indicating the key source.
- **key** (Bytearray) – key of the device to register.

Returns

the register device operation status or None if the answer received is not a RegisterDeviceStatusPacket.

Return type ZigbeeRegisterStatus

Raises

- **TimeoutException** – if the answer is not received in the configured timeout.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – if the XBee device’s serial port is closed.
- **ValueError** – if registrant_address is None or if options is None.

See also:

RegisterKeyOptions
XBee64BitAddress
ZigbeeRegisterStatus

register_joining_device_async (registrant_address, options, key)
Securely registers a joining device to a trust center. Registration is the process by which a node is authorized to join the network using a preconfigured link key or installation code that is conveyed to the trust center out-of-band (using a physical interface and not over-the-air).

This method is asynchronous, which means that it will not wait for an answer after sending the register frame.

Parameters

- **registrant_address** (XBee64BitAddress) – the 64-bit address of the device to register.
- **options** (RegisterKeyOptions) – the register options indicating the key source.
- **key** (Bytearray) – key of the device to register.

Raises

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **XBeeException** – if the XBee device’s serial port is closed.

• **ValueError** – if `registrant_address` is `None` or if `options` is `None`.

See also:

RegisterKeyOptions
XBee64BitAddress

unregister_joining_device(*unregistrant_address*)

Unregisters a joining device from a trust center.

This method is synchronous, it sends the unregister joining device packet and waits for the answer of the operation. Then, returns the corresponding status.

Parameters `unregistrant_address` (*XBee64BitAddress*) – the 64-bit address of the device to unregister.

Returns the unregister device operation status or `None` if the answer received is not a `RegisterDeviceStatusPacket`.

Return type `ZigbeeRegisterStatus`

Raises

• **TimeoutException** – if the answer is not received in the configured timeout.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if the XBee device’s serial port is closed.

• **ValueError** – if `registrant_address` is `None`.

See also:

XBee64BitAddress
ZigbeeRegisterStatus

unregister_joining_device_async(*unregistrant_address*)

Unregisters a joining device from a trust center.

This method is asynchronous, which means that it will not wait for an answer after sending the unregister frame.

Parameters `unregistrant_address` (*XBee64BitAddress*) – the 64-bit address of the device to unregister.

Raises

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if the XBee device’s serial port is closed.

• **ValueError** – if `registrant_address` is `None`.

See also:
XBee64BitAddress

**get_routes** *(route_callback=None, process_finished_callback=None, timeout=None)*

Returns the routes of this XBee. If route_callback is not defined, the process blocks until the complete routing table is read.

**Parameters**

- **route_callback** *(Function, optional, default=``None``)* – method called when a new route is received. Receives two arguments:
  - The XBee that owns this new route.
  - The new route.
- **process_finished_callback** *(Function, optional, default=``None``)* – method to execute when the process finishes. Receives two arguments:
  - The XBee device that executed the ZDO command.
  - A list with the discovered routes.
  - An error message if something went wrong.
- **timeout** *(Float, optional, default=``RouteTableReader.DEFAULT_TIMEOUT``)* – The ZDO command timeout in seconds.

**Returns**

List of **com.digi.models.zdo.Route** when route_callback is defined, None otherwise (in this case routes are received in the callback).

**Return type** List

**Raises**

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **OperationNotSupportedException** – If XBee protocol is not Zigbee or Smart Energy.
- **XBeeException** – If the XBee device’s serial port is closed.

**See also:**

**com.digi.models.zdo.Route**

**get_neighbors** *(neighbor_callback=None, process_finished_callback=None, timeout=None)*

Returns the neighbors of this XBee. If neighbor_callback is not defined, the process blocks until the complete neighbor table is read.

**Parameters**

- **neighbor_callback** *(Function, optional, default=``None``)* – method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
• **process_finished_callback** *(Function, optional, default=``None``)* — method to execute when the process finishes. Receives two arguments:
  - The XBee device that executed the ZDO command.
  - A list with the discovered neighbors.
  - An error message if something went wrong.

• **timeout** *(Float, optional, default=``NeighborTableReader.DEFAULT_TIMEOUT``)* — The ZDO command timeout in seconds.

Returns

List of *com.digi.models.zdo.Neighbor* when *neighbor_callback* is defined, *None* otherwise (in this case neighbors are received in the callback).

Return type  List

Raises **OperationNotSupportedException** – If XBee protocol is not Zigbee or Smart Energy.

See also:

*com.digi.models.zdo.Neighbor*

**add_bluetooth_data_received_callback** *(callback)*
Adds a callback for the event *digi.xbee.reader.BluetoothDataReceived*.

Parameters **callback** *(Function)* — the callback. Receives one argument.

• The Bluetooth data as a Bytearray

**add_data_received_callback** *(callback)*
Adds a callback for the event *digi.xbee.reader.DataReceived*.

Parameters **callback** *(Function)* — the callback. Receives one argument.

• The data received as an *digi.xbee.models.message.XBeeMessage*

**add_expl_data_received_callback** *(callback)*
Adds a callback for the event *digi.xbee.reader.ExplicitDataReceived*.

Parameters **callback** *(Function)* — the callback. Receives one argument.

• The explicit data received as a *digi.xbee.models.message.ExplicitXBeeMessage*

**add_io_sample_received_callback** *(callback)*
Adds a callback for the event *digi.xbee.reader.IOSampleReceived*.

Parameters **callback** *(Function)* — the callback. Receives three arguments.

• The received IO sample as an *digi.xbee.io.IOSample*
• The remote XBee device who has sent the packet as a *RemoteXBeeDevice*
• The time in which the packet was received as an Integer

**add_micropython_data_received_callback** *(callback)*
Adds a callback for the event *digi.xbee.reader.MicroPythonDataReceived*.

Parameters **callback** *(Function)* — the callback. Receives one argument.
• The MicroPython data as a Bytearray

add_modem_status_received_callback(callback)
    Adds a callback for the event `digi.xbee.reader.ModemStatusReceived`.

    Parameters: callback (Function) – the callback. Receives one argument.
    • The modem status as a `digi.xbee.models.status.ModemStatus`

add_packet_received_callback(callback)
    Adds a callback for the event `digi.xbee.reader.PacketReceived`.

    Parameters: callback (Function) – the callback. Receives two arguments.
    • The received packet as a `digi.xbee.packets.base.XBeeAPIPacket`

add_route_received_callback(callback)
    Adds a callback for the event `RouteReceived`. This works for Zigbee and Digimesh devices.

    Parameters: callback (Function) – the callback. Receives three arguments.
    • source (XBeeDevice): The source node.
    • destination (RemoteXBeeDevice): The destination node.
    • hops (List): List of intermediate hops from closest to source to closest to destination (RemoteXBeeDevice).

    See also:

    `XBeeDevice.del_route_received_callback()`

add_socket_data_received_callback(callback)
    Adds a callback for the event `digi.xbee.reader.SocketDataReceived`.

    Parameters: callback (Function) – the callback. Receives two arguments.
    • The socket ID as an Integer.
    • The data received as Bytearray

add_socket_data_received_from_callback(callback)
    Adds a callback for the event `digi.xbee.reader.SocketDataReceivedFrom`.

    Parameters: callback (Function) – the callback. Receives three arguments.
    • The socket ID as an Integer.
    • A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
    • The data received as Bytearray

add_socket_state_received_callback(callback)
    Adds a callback for the event `digi.xbee.reader.SocketStateReceived`.

    Parameters: callback (Function) – the callback. Receives two arguments.
    • The socket ID as an Integer.
    • The state received as a `SocketState`

add_user_data_relay_received_callback(callback)
    Adds a callback for the event `digi.xbee.reader.RelayDataReceived`. 
**Parameters** `callback` (*Function*) – the callback. Receives one argument.

- The relay data as a `digi.xbee.models.message.UserDataRelayMessage`

**apply_changes()**

Applies changes via AC command.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**apply_profile** *(profile_path, progress_callback=None)*

Applies the given XBe profile to the XBe device.

**Parameters**

- `profile_path` (*String*) – path of the XBe profile file to apply.
- `progress_callback` (*Function, optional*) – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

**Raises**

- `XBeeException` – if the device is not open.
- `InvalidOperatingModeException` – if the device operating mode is invalid.
- `UpdateProfileException` – if there is any error applying the XBe profile.
- `OperationNotSupportedException` – if XBe profiles are not supported in the XBe device.

**close()**

Closes the communication with the XBe device.

This method guarantees that all threads running are stopped and the serial port is closed.

**comm_iface**

Returns the hardware interface associated to the XBe device.

**Returns** the hardware interface associated to the XBe device.

**Return type** `XBeeCommunicationInterface`

**See also:**

`XBeeSerialPort`

**classmethod create_xbee_device** *(comm_port_data)*

Creates and returns an `XBeeDevice` from data of the port to which it is connected.

**Parameters**
• **comm_port_data** *(Dictionary)* – dictionary with all comm port data needed.

• **dictionary keys are** *(The)* –
  
  “baudRate” –> Baud rate.
  ”port” –> Port number.
  ”bitSize” –> Bit size.
  ”stopBits” –> Stop bits.
  ”parity” –> Parity.
  ”flowControl” –> Flow control.
  ”timeout” for –> Timeout for synchronous operations (in seconds).

Returns the XBee device created.

Return type **XBeeDevice**

Raises **SerialException** – if the port you want to open does not exist or is already opened.

See also:

**XBeeDevice**

delete (**del**) **bluetooth_data_received_callback** *(callback)*

Deletes a callback for the callback list of **digi.xbee.reader.BluetoothDataReceived** event.

Parameters **callback** *(Function)* – the callback to delete.

Raises **ValueError** – if **callback** is not in the callback list of **digi.xbee.reader.BluetoothDataReceived** event.

delete (**del**) **data_received_callback** *(callback)*

Deletes a callback for the callback list of **digi.xbee.reader.DataReceived** event.

Parameters **callback** *(Function)* – the callback to delete.

Raises **ValueError** – if **callback** is not in the callback list of **digi.xbee.reader.DataReceived** event.

delete (**del**) **expl_data_received_callback** *(callback)*

Deletes a callback for the callback list of **digi.xbee.reader.ExplicitDataReceived** event.

Parameters **callback** *(Function)* – the callback to delete.

Raises **ValueError** – if **callback** is not in the callback list of **digi.xbee.reader.ExplicitDataReceived** event.

delete (**del**) **io_sample_received_callback** *(callback)*

Deletes a callback for the callback list of **digi.xbee.reader.IOSampleReceived** event.

Parameters **callback** *(Function)* – the callback to delete.

Raises **ValueError** – if **callback** is not in the callback list of **digi.xbee.reader.IOSampleReceived** event.

delete (**del**) **micropython_data_received_callback** *(callback)*

Deletes a callback for the callback list of **digi.xbee.reader.MicroPythonDataReceived** event.

Parameters **callback** *(Function)* – the callback to delete.
Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

```python
del_modem_status_received_callback(callback)
```
Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

Parameters `callback` (Function) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

```python
del_packet_received_callback(callback)
```
Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

Parameters `callback` (Function) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.PacketReceived` event.

```python
del_route_received_callback(callback)
```
Deletes a callback for the callback list of `RouteReceived` event.

Parameters `callback` (Function) – the callback to delete.

See also:

`XBeeDevice.add_route_received_callback()`

```python
del_socket_data_received_callback(callback)
```
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

Parameters `callback` (Function) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.

```python
del_socket_data_received_from_callback(callback)
```
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

Parameters `callback` (Function) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

```python
del_socket_state_received_callback(callback)
```
Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

Parameters `callback` (Function) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

```python
del_user_data_relay_received_callback(callback)
```
Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

Parameters `callback` (Function) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.

```python
disable_bluetooth()
```
Disables the Bluetooth interface of this XBee device.
Note that your device must have Bluetooth Low Energy support to use this method.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**enable_apply_changes**(value)

Sets the apply_changes flag.

**Parameters**

value (Boolean) – True to enable the apply changes flag, False to disable it.

**enable_bluetooth()**

Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**execute_command**(parameter)

Executes the provided command.

**Parameters**

parameter (String) – The name of the AT command to be executed.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**flush_queues()**

Flushes the packets queue.

**get_16bit_addr()**

Returns the 16-bit address of the XBee device.

**Returns** the 16-bit address of the XBee device.

**Return type** `XBee16BitAddress`

See also:

`XBee16BitAddress`
**get_64bit_addr()**

Returns the 64-bit address of the XBee device.

**Returns** the 64-bit address of the XBee device.

**Return type** XBee64BitAddress

See also:

XBee64BitAddress

**get_adc_value(io_line)**

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.set_io_configuration() and IOMode.ADC.

**Parameters** io_line (IOLine) – the IO line to get its ADC value.

**Returns** the analog value corresponding to the provided IO line.

**Return type** Integer

**Raises**

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.
- *OperationNotSupportedException* – if the response does not contain the value for the given IO line.

See also:

IOLine

**get_api_output_mode()**

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

**Returns** the API output mode of the XBee device.

**Return type** APIOutputMode

**Raises**

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• *ATCommandException* – if the response is not as expected.

See also:

```
APIOutputMode
```

### get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

- **Returns** the parameter value.
- **Return type** Bytearray
- **Raises**
  - *TimeoutException* – if the response is not received before the read timeout expires.
  - *XBeeException* – if the XBee device’s serial port is closed.
  - *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  - *ATCommandException* – if the response is not as expected.
  - *OperationNotSupportedException* – if it is not supported by the current protocol.

See also:

```
digi.xbee.models.mode.APIOutputModeBit
```

### get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

- **Returns** The Bluetooth MAC address.
- **Return type** String
- **Raises**
  - *TimeoutException* – if the response is not received before the read timeout expires.
  - *XBeeException* – if the XBee device’s serial port is closed.
  - *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

### get_current_frame_id()

Returns the last used frame ID.

- **Returns** the last used frame ID.
- **Return type** Integer
**`get_dest_address()`**  
Returns the 64-bit address of the XBee device that data will be reported to.

- **Returns** the address.
- **Return type** `XBee64BitAddress`
- **Raises** `TimeoutException` – if the response is not received before the read timeout expires.

See also:

`XBee64BitAddress`

**`get_dio_value(io_line)`**  
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

- **Parameters** `io_line` (IOLine) – the DIO line to get its digital value.
- **Returns** current value of the provided IO line.
- **Return type** `IOValue`
- **Raises**
  - `TimeoutException` – if the response is not received before the read timeout expires.
  - `XBeeException` – if the XBee device’s serial port is closed.
  - `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  - `ATCommandException` – if the response is not as expected.
  - `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

`IOLine`

`IOValue`

**`get_firmware_version()`**  
Returns the firmware version of the XBee device.

- **Returns** the hardware version of the XBee device.
- **Return type** `Bytearray`

**`get_hardware_version()`**  
Returns the hardware version of the XBee device.

- **Returns** the hardware version of the XBee device.
- **Return type** `HardwareVersion`

See also:
HardwareVersion

get_io_configuration(io_line)
    Returns the configuration of the provided IO line.
    Parameters io_line (IOLine) – the io line to configure.
    Returns the IO mode of the IO line provided.
    Return type IOMode
    Raises
        • TimeoutException – if the response is not received before the read timeout expires.
        • XBeeException – if the XBee device’s serial port is closed.
        • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
        • ATCommandException – if the response is not as expected.
        • OperationNotSupportedException – if the received data is not an IO mode.

get_io_sampling_rate()
    Returns the IO sampling rate of the XBee device.
    Returns the IO sampling rate of XBee device.
    Return type Integer
    Raises
        • TimeoutException – if the response is not received before the read timeout expires.
        • XBeeException – if the XBee device’s serial port is closed.
        • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
        • ATCommandException – if the response is not as expected.

get_network()
    Returns this XBee device’s current network.
    Returns XBeeDevice.XBeeNetwork

get_next_frame_id()
    Returns the next frame ID of the XBee device.
    Returns The next frame ID of the XBee device.
    Return type Integer

get_node_id()
    Returns the Node Identifier (NI) value of the XBee device.
    Returns the Node Identifier (NI) of the XBe device.
    Return type String

get_pan_id()
    Returns the operating PAN ID of the XBee device.
    Returns operating PAN ID of the XBee device.
    Return type Bytearray
Raises `TimeoutException` – if the response is not received before the read timeout expires.

```python
get_parameter(param, parameter_value=None)
```
Override.

See also:

```
AbstractXBeeDevice.get_parameter()
```

```python
get_power_level()
```
Returns the power level of the XBee device.

- **Returns**: the power level of the XBee device.
- **Return type**: `PowerLevel`
- **Raises**: `TimeoutException` – if the response is not received before the read timeout expires.

See also:

```
PowerLevel
```

```python
get_pwm_duty_cycle(io_line)
```
Returns the PWM duty cycle in % corresponding to the provided IO line.

- **Parameters**: `io_line` (*IOLine*) – the IO line to get its PWM duty cycle.
- **Returns**: the PWM duty cycle of the given IO line or `None` if the response is empty.
- **Return type**: `Integer`
- **Raises**:
  - `TimeoutException` – if the response is not received before the read timeout expires.
  - `XBeeException` – if the XBee device’s serial port is closed.
  - `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  - `ATCommandException` – if the response is not as expected.
  - `ValueError` – if the passed `IO_LINE` has no PWM capability.

See also:

```
IOLine
```

```python
get_role()
```
Gets the XBee role.

- **Returns**: the role of the XBee.
- **Return type**: `digi.xbee.models.protocol.Role`

See also:
digixbee.models.protocol.Role

get_route_to_node(remote, timeout=10)

Gets the route from this XBee to the given remote node.

Parameters

• remote (RemoteXBeeDevice) – The remote node.

• timeout (Float, optional, default=10) – Maximum number of seconds to wait for the route.

Returns

Tuple containing route data (None if the route was not read in the provided timeout):
- source (RemoteXBeeDevice): The source node of the route. - destination (RemoteXBeeDevice): The destination node of the route.

• hops (List): List of intermediate nodes (RemoteXBeeDevice) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout()

Returns the serial port read timeout.

Returns the serial port read timeout in seconds.

Return type Integer

get_xbee_device_callbacks()

Returns this XBee internal callbacks for process received packets.

This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

Returns PacketReceived

has_explicit_packets()

Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

Returns True if this XBee device’s queue has explicit packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()

Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

Returns True if this XBee device’s queue has packets, False otherwise.

Return type Boolean

See also:
XBeeDevice.has_explicit_packets()

**is_apply_changes_enabled()**

Returns whether the apply_changes flag is enabled or not.

Returns **True** if the apply_changes flag is enabled, **False** otherwise.

Return type Boolean

**is_open()**

Returns whether this XBee device is open or not.

Returns **Boolean. True** if this XBee device is open, **False** otherwise.

**is_remote()**

Override method.

See also:

AbstractXBeeDevice.is_remote()

**log**

Returns the XBee device log.

Returns the XBee device logger.

Return type Logger

**operating_mode**

Returns this XBee device’s operating mode.

Returns **OperatingMode. This XBee device’s operating mode.**

**reachable**

Returns whether the XBee is reachable.

Returns **True** if the device is reachable, **False** otherwise.

Return type Boolean

**read_data**(timeout=None)

Reads new data received by this XBee device.

If a timeout is specified, this method blocks until new data is received or the timeout expires, throwing in that case a **TimeoutException**.

**Parameters**

- **timeout**(Integer, optional) – read timeout in seconds. If it’s **None**, this method is non-blocking and will return **None** if there is no data available.

**Returns**

the read message or **None** if this XBee did not receive new data.

Return type XBeeMessage

**Raises**

- **ValueError** – if a timeout is specified and is less than 0.

- **TimeoutException** – if a timeout is specified and no data was received during that time.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **XBeeException** – if the XBee device’s serial port is closed.

**See also:**

**XBeeMessage**

**read_data_from** *(remote_xbee_device, timeout=None)*

Reads new data received from the given remote XBee device.

If a **timeout** is specified, this method blocks until new data is received or the timeout expires, throwing in that case a **TimeoutException**.

**Parameters**

- **remote_xbee_device** *(RemoteXBeeDevice)* – the remote XBee device that sent the data.
- **timeout** *(Integer, optional) – read timeout in seconds. If it’s None, this method is non-blocking and will return None if there is no data available.*

**Returns**

the read message sent by **remote_xbee_device** or **None** if this XBee did not receive new data.

**Return type** **XBeeMessage**

**Raises**

- **ValueError** – if a timeout is specified and is less than 0.
- **TimeoutException** – if a timeout is specified and no data was received during that time.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – if the XBee device’s serial port is closed.

**See also:**

**XBeeMessage**

**RemoteXBeeDevice**

**read_device_info** *(init=True, fire_event=True)*

Updates all instance parameters reading them from the XBee device.

**Parameters**

- **init** *(Boolean, optional, default=True)* – If **False** only not initialized parameters are read, all if **True**.
- **fire_event** *(Boolean, optional, default=True)* – **True** to launch and update event if any parameter changed, **False** otherwise.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

**read_io_sample()**

Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

**Returns**  the IO sample read from the XBee device.

**Return type**  IOSample

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

**See also:**

IOSample

**reset()**

Override method.

**See also:**

AbstractXBeeDevice.reset()

**scan_counter**

Returns the scan counter for this node.

**Returns**  The scan counter for this node.

**Return type**  Integer

**send_bluetooth_data**(data)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

**Parameters**  data (Bytearray) – Data to send.

**Raises**

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if there is any problem sending the data.

**See also:**

XBeeDevice.send_micropython_data()
XBeeDevice.send_user_data_relay()
send_data (remote_xbee_device, data, transmit_options=0)
Block method. This method sends data to a remote XBee device synchronously.
This method will wait for the packet response.
The default timeout for this method is XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS.

Parameters

• remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to send
data to.
• data (String or Bytearray) – the raw data to send.
• transmit_options (Integer, optional) – transmit options, bitfield of
  TransmitOptions. Default to TransmitOptions.NONE.value.

Returns XBeePacket the response.

Raises

• ValueError – if remote_xbee_device is None.
• TimeoutException – if this method can’t read a response packet in XBeeDevice.
  _DEFAULT_TIMEOUT_SYNC_OPERATIONS seconds.
• InvalidOperatingModeException – if the XBee device’s operating mode is not
  API or ESCAPED API. This method only checks the cached value of the operating mode.
• TransmitException – if the status of the response received is not OK.
• XBeeException – if the XBee device’s serial port is closed.

See also:

RemoteXBeeDevice
XBeePacket

send_data_async (remote_xbee_device, data, transmit_options=0)
Non-blocking method. This method sends data to a remote XBee device.
This method won’t wait for the response.

Parameters

• remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to send
data to.
• data (String or Bytearray) – the raw data to send.
• transmit_options (Integer, optional) – transmit options, bitfield of
  TransmitOptions. Default to TransmitOptions.NONE.value.

Raises

• InvalidOperatingModeException – if the XBee device’s operating mode is not
  API or ESCAPED API. This method only checks the cached value of the operating mode.
• XBeeException – if the XBee device’s serial port is closed.

See also:
RemoteXBeeDevice

**send_data_broadcast** *(data, transmit_options=0)*

Sends the provided data to all the XBee nodes of the network (broadcast).

This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using the AbstractXBeeDevice.set_sync_ops_timeout() method and can be consulted with AbstractXBeeDevice.get_sync_ops_timeout() method.

**Parameters**

- **data** *(String or bytearray)* – data to send.
- **transmit_options** *(Integer, optional)* – transmit options, bitfield of TransmitOptions. Default to TransmitOptions.NONE.value.

**Raises**

- **TimeoutException** – if this method can’t read a response packet in XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS seconds.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **TransmitException** – if the status of the response received is not OK.
- **XBeeException** – if the XBee device’s serial port is closed.

**send_micropython_data** *(data)*

Sends the given data to the MicroPython interface using a User Data Relay frame.

**Parameters**

- **data** *(Bytearray)* – Data to send.

**Raises**

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – if there is any problem sending the data.

**See also:**

XBeeDevice.send_bluetooth_data()
XBeeDevice.send_user_data_relay()

**send_packet** *(packet, sync=False)*

Override method.

**See also:**

AbstractXBeeDevice._send_packet()

**send_packet_sync_and_get_response** *(packet_to_send, timeout=None)*

Override method.

**See also:**
AbstractXBeeDevice._send_packet_sync_and_get_response()

send_user_data_relay(local_interface, data)
Sends the given data to the given XBee local interface.

Parameters

• local_interface (XBeeLocalInterface) – Destination XBee local interface.
• data (Bytearray) – Data to send.

Raises

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ValueError – if local_interface is None.
• XBeeException – if there is any problem sending the User Data Relay.

See also:

XBeeLocalInterface

serial_port
Returns the serial port associated to the XBee device, if any.

Returns

the serial port associated to the XBee device. Returns None if the local XBee does not use serial communication.

Return type XBeeSerialPort

See also:

XBeeSerialPort

set_16bit_addr(value)
Sets the 16-bit address of the XBee device.

Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is not 802.15.4.
**set_api_output_mode** *(api_output_mode)*

Deprecated since version 1.3: Use **set_api_output_mode_value()**

Sets the API output mode of the XBee device.

**Parameters**

`api_output_mode` *(APIOutputMode)* – the new API output mode of the XBee device.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the current protocol is ZigBee

**See also:**

APIOutputMode

**set_api_output_mode_value** *(api_output_mode)*

Sets the API output mode of the XBee.

**Parameters**

`api_output_mode` *(Integer)* – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

**See also:**

digi.xbee.models.mode.APIOutputModeBit

**set_dest_address** *(addr)*

Sets the 64-bit address of the XBee device that data will be reported to.

**Parameters**

`addr` *(XBee64BitAddress or RemoteXBeeDevice)* – the address itself or the remote XBee device that you want to set up its address as destination address.

**Raises**

- `TimeoutException` – If the response is not received before the read timeout expires.
• **XBeeException** – If the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – If the response is not as expected.

• **ValueError** – If `addr` is None.

**set_dio_change_detection**(*io_lines_set*)
Sets the digital IO lines to be monitored and sampled whenever their status changes.

A None set of lines disables this feature.

**Parameters**

- `io_lines_set` – set of IOLine.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.

See also:

**IOLine**

**set_dio_value**(*io_line*, *io_value*)
Sets the digital value (high or low) to the provided IO line.

**Parameters**

- `io_line` (*IOLine*) – the digital IO line to sets its value.

- `io_value` (*IOValue*) – the IO value to set to the IO line.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.

See also:

**IOLine**  
**IOValue**

**set_io_configuration**(*io_line*, *io_mode*)
Sets the configuration of the provided IO line.

**Parameters**

...
set_io_line\( (\text{IOLine}) \) – the IO line to configure.

set_io_mode\( (\text{IOMode}) \) – the IO mode to set to the IO line.

 Raises

- \text{TimeoutException} – if the response is not received before the read timeout expires.
- \text{XBeeException} – if the XBe device’s serial port is closed.
- \text{InvalidOperatingModeException} – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \text{ATCommandException} – if the response is not as expected.

See also:

\text{IOLine}
\text{IOMode}

\text{set_io_sampling_rate}(\text{rate})

Sets the IO sampling rate of the XBe device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

 Parameters \text{rate}(\text{Integer}) – the new IO sampling rate of the XBe device in seconds.

 Raises

- \text{TimeoutException} – if the response is not received before the read timeout expires.
- \text{XBeeException} – if the XBe device’s serial port is closed.
- \text{InvalidOperatingModeException} – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \text{ATCommandException} – if the response is not as expected.

\text{set_node_id}(\text{node_id})

Sets the Node Identifier (NI) value of the XBe device..

 Parameters \text{node_id}(\text{String}) – the new Node Identifier (NI) of the XBe device.

 Raises

- \text{ValueError} – if \text{node_id} is None or its length is greater than 20.
- \text{TimeoutException} – if the response is not received before the read timeout expires.

\text{set_pan_id}(\text{value})

Sets the operating PAN ID of the XBe device.

 Parameters \text{value}(\text{Bytearray}) – the new operating PAN ID of the XBe device. Must have only 1 or 2 bytes.

 Raises \text{TimeoutException} – if the response is not received before the read timeout expires.

\text{set_parameter}(\text{param}, \text{value})

Override.

 See: \text{AbstractXBeeDevice.set_parameter()}

\text{set_power_level}(\text{power_level})

Sets the power level of the XBe device.
Parameters **power_level** (*PowerLevel*) – the new power level of the XBee device.

Raises **TimeoutException** – if the response is not received before the read timeout expires.

See also:

*PowerLevel*

**set_pwm_duty_cycle** (*io_line, cycle*)
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

- **io_line** (*IOLine*) – the IO Line to be assigned.
- **cycle** (*Integer*) – duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **ValueError** – if the given IO line does not have PWM capability or **cycle** is not between 0 and 100.

See also:

*IOLine*

*IOMode.PWM*

**set_sync_ops_timeout** (*sync_ops_timeout*)
Sets the serial port read timeout.

Parameters **sync_ops_timeout** (*Integer*) – the read timeout, expressed in seconds.

**update_bluetooth_password** (*new_password*)
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
**update_device_data_from**(device)

Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**

- **device** (<AbstractXBeeDevice>) – the XBee device to get the data from.

**Returns**

- True if the device data has been updated, False otherwise.

**Return type**

Boolean

**update_firmware**(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)

Performs a firmware update operation of the device.

**Parameters**

- **xml_firmware_file** (String) – path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (String, optional) – location of the XBee binary firmware file.
- **bootloader_firmware_file** (String, optional) – location of the bootloader binary firmware file.
- **timeout** (Integer, optional) – the maximum time to wait for target read operations during the update process.
- **progress_callback** (Function, optional) – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

**Raises**

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.
- **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes**()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method **is_apply_configuration_changes_enabled()** to get its status and **enable_apply_configuration_changes()** to enable/disable the option. If it is disabled, method **apply_changes()** can be used in order to manually apply the changes.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

```python
class digi.xbee.devices.IPDevice(port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)
```

**Bases:** `digi.xbee.devices.XBeeDevice`

This class provides common functionality for XBee IP devices.

Class constructor. Instantiates a new IPDevice with the provided parameters.

**Parameters**

- **port** *(Integer or String)* – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. `/dev/ttyUSB0` on ‘GNU/Linux’ or ‘COM3’ on Windows.

- **baud_rate** *(Integer)* – the serial port baud rate.

- **data_bits** *(Integer, default (_sync_ops_timeout) – serial.EIGHTBITS)*: comm port bitsize.

- **stop_bits** *(Integer, default – serial.STOPBITS_ONE)*: comm port stop bits.

- **parity** *(Character, default (parity) – serial.PARITY_NONE)*: comm port parity.

- **flow_control** *(Integer, default – FlowControl.NONE)*: comm port flow control.

- **comm_iface** *(XBeeCommunicationInterface)* – the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor.: 

**See also:**

- XBeeDevice
- XBeeDevice.__init__()

```python
read_device_info(init=True, fire_event=True)
```

Override.

**See also:**

- AbstractXBeeDevice.read_device_info()

```python
get_ip_addr()
```

Returns the IP address of this IP device.

To refresh this value use the method **IPDevice.read_device_info()**.
Returns  The IP address of this IP device.

Return type  `ipaddress.IPv4Address`

See also:

`ipaddress.IPv4Address`

`set_dest_ip_addr` *(address)*

Sets the destination IP address.

Parameters  `address` *(ipaddress.IPv4Address)* – Destination IP address.

Raises

- `ValueError` – if `address` is `None`.
- `TimeoutException` – if there is a timeout setting the destination IP address.
- `XBeeException` – if there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

`get_dest_ip_addr` *(*)

Returns the destination IP address.

Returns  The configured destination IP address.

Return type  `ipaddress.IPv4Address`

Raises

- `TimeoutException` – if there is a timeout getting the destination IP address.
- `XBeeException` – if there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

`add_ip_data_received_callback` *(callback)*

Adds a callback for the event `digi.xbee.reader.IPDataReceived`.

Parameters  `callback` *(Function)* – the callback. Receives one argument.

- The data received as an `digi.xbee.models.message.IPMessage`

`del_ip_data_received_callback` *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.IPDataReceived` event.

Parameters  `callback` *(Function)* – the callback to delete.

Raises  `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.IPDataReceived` event.
start_listening(source_port)
Starts listening for incoming IP transmissions in the provided port.

Parameters
source_port (Integer) – Port to listen for incoming transmissions.

Raises
- ValueError – if source_port is less than 0 or greater than 65535.
- TimeoutException – if there is a timeout setting the source port.
- XBeeException – if there is any other XBee related exception.

stop_listening()
Stops listening for incoming IP transmissions.

Raises
- TimeoutException – if there is a timeout processing the operation.
- XBeeException – if there is any other XBee related exception.

send_ip_data(ip_addr, dest_port, protocol, data, close_socket=False)
Sends the provided IP data to the given IP address and port using the specified IP protocol. For TCP and TCP SSL protocols, you can also indicate if the socket should be closed when data is sent.

This method blocks till a success or error response arrives or the configured receive timeout expires.

Parameters
- ip_addr (ipaddress.IPv4Address) – The IP address to send IP data to.
- dest_port (Integer) – The destination port of the transmission.
- protocol (IPProtocol) – The IP protocol used for the transmission.
- data (String or bytearray) – The IP data to be sent.
- close_socket (Boolean, optional) – True to close the socket just after the transmission. False to keep it open. Default to False.

Raises
- ValueError – if ip_addr is None.
- ValueError – if protocol is None.
- ValueError – if data is None.
- ValueError – if dest_port is less than 0 or greater than 65535.
- OperationNotSupportedException – if the device is remote.
- TimeoutException – if there is a timeout sending the data.
- XBeeException – if there is any other XBee related exception.

send_ip_data_async(ip_addr, dest_port, protocol, data, close_socket=False)
Sends the provided IP data to the given IP address and port asynchronously using the specified IP protocol. For TCP and TCP SSL protocols, you can also indicate if the socket should be closed when data is sent.

Asynchronous transmissions do not wait for answer from the remote device or for transmit status packet.

Parameters
- ip_addr (ipaddress.IPv4Address) – The IP address to send IP data to.
- dest_port (Integer) – The destination port of the transmission.
• **protocol** *(IPProtocol)* – The IP protocol used for the transmission.

• **data** *(String or Bytearray)* – The IP data to be sent.

• **close_socket** *(Boolean, optional)* – True to close the socket just after the transmission. False to keep it open. Default to False.

**Raises**

• `ValueError` – if `ip_addr` is `None`.

• `ValueError` – if `protocol` is `None`.

• `ValueError` – if `data` is `None`.

• `ValueError` – if `dest_port` is less than 0 or greater than 65535.

• `OperationNotSupportedException` – if the device is remote.

• `XBeeException` – if there is any other XBee related exception.

**send_ip_data_broadcast** *(dest_port, data)*

Sends the provided IP data to all clients.

This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

**Parameters**

• **dest_port** *(Integer)* – The destination port of the transmission.

• **data** *(String or Bytearray)* – The IP data to be sent.

**Raises**

• `ValueError` – if `data` is `None`.

• `ValueError` – if `dest_port` is less than 0 or greater than 65535.

• `TimeoutException` – if there is a timeout sending the data.

• `XBeeException` – if there is any other XBee related exception.

**read_ip_data** *(timeout=3)*

Reads new IP data received by this XBee device during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

**Parameters** `timeout` *(Integer, optional)* – The time to wait for new IP data in seconds.

**Returns** IP message, `None` if this device did not receive new data.

**Return type** `IPMessage`

**Raises** `ValueError` – if `timeout` is less than 0.

**read_ip_data_from** *(ip_addr, timeout=3)*

Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.
For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

**Parameters**

- `ip_addr` (:class:`ipaddress.IPv4Address`) – The IP address to read data from.
- `timeout` (:class:`int`, optional) – The time to wait for new IP data in seconds.

**Returns**

- **IP message**, `None` if this device did not receive new data from the provided IP address.

**Return type** `IPMessage`

**Raises** `ValueError` – if `timeout` is less than 0.

### get_network()

Deprecated.

This protocol does not support the network functionality.

### get_16bit_addr()

Deprecated.

This protocol does not have an associated 16-bit address.

### get_dest_address()

Deprecated.

Operation not supported in this protocol. Use `IPDevice.get_dest_ip_addr()` instead. This method will raise an `AttributeError`.

### set_dest_address(addr)

Deprecated.

Operation not supported in this protocol. Use `IPDevice.set_dest_ip_addr()` instead. This method will raise an `AttributeError`.

### get_pan_id()

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

### set_pan_id(value)

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

### add_data_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

### del_data_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

### add_expl_data_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

472 Chapter 2. Contents
del_expl_data_received_callback (callback)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

read_data (timeout=None, explicit=False)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

read_data_from (remote_xbee_device, timeout=None, explicit=False)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

send_data_broadcast (data, transmit_options=0)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

send_data (remote_xbee_device, data, transmit_options=0)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

send_data_async (remote_xbee_device, data, transmit_options=0)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

add_bluetooth_data_received_callback (callback)
Adds a callback for the event digi.xbee.reader.BluetoothDataReceived.

Parameters callback (Function) – the callback. Receives one argument.
• The Bluetooth data as a Bytearray

add_io_sample_received_callback (callback)
Adds a callback for the event digi.xbee.reader.IOSampleReceived.

Parameters callback (Function) – the callback. Receives three arguments.
• The received IO sample as an digi.xbee.io.IOSample
• The remote XBee device who has sent the packet as a RemoteXBeeDevice
• The time in which the packet was received as an Integer

add_micropython_data_received_callback (callback)
Adds a callback for the event digi.xbee.reader.MicroPythonDataReceived.

Parameters callback (Function) – the callback. Receives one argument.
• The MicroPython data as a Bytearray

add_modem_status_received_callback (callback)
Adds a callback for the event digi.xbee.reader.ModemStatusReceived.

Parameters callback (Function) – the callback. Receives one argument.
• The modem status as a digi.xbee.models.status.ModemStatus

add_packet_received_callback (callback)
Adds a callback for the event digi.xbee.reader.PacketReceived.

Parameters callback (Function) – the callback. Receives two arguments.
• The received packet as a digi.xbee.packets.base.XBeeAPIPacket
add_route_received_callback (callback)
Adds a callback for the event RouteReceived. This works for Zigbee and Digimesh devices.

Parameters callback (Function) – the callback. Receives three arguments.
- source (XBeeDevice): The source node.
- destination (RemoteXBeeDevice): The destination node.
- hops (List): List of intermediate hops from closest to source to closest to destination (RemoteXBeeDevice).

See also:
XBeeDevice.del_route_received_callback()

add_socket_data_received_callback (callback)
Adds a callback for the event digi.xbee.reader.SocketDataReceived.

Parameters callback (Function) – the callback. Receives two arguments.
- The socket ID as an Integer.
- The data received as Bytearray

add_socket_data_received_from_callback (callback)
Adds a callback for the event digi.xbee.reader.SocketDataReceivedFrom.

Parameters callback (Function) – the callback. Receives three arguments.
- The socket ID as an Integer.
- A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
- The data received as Bytearray

add_socket_state_received_callback (callback)
Adds a callback for the event digi.xbee.reader.SocketStateReceived.

Parameters callback (Function) – the callback. Receives two arguments.
- The socket ID as an Integer.
- The state received as a SocketState

add_user_data_relay_received_callback (callback)
Adds a callback for the event digi.xbee.reader.RelayDataReceived.

Parameters callback (Function) – the callback. Receives one argument.
- The relay data as a digi.xbee.models.message.UserDataRelayMessage

apply_changes ()
Applies changes via AC command.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

### apply_profile (profile_path, progress_callback=None)
Applies the given XBee profile to the XBee device.

**Parameters**
- **profile_path** (*String*) – path of the XBee profile file to apply.
- **progress_callback** (*Function, optional*) –
  function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

**Raises**
- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **UpdateProfileException** – if there is any error applying the XBee profile.
- **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

### close ()
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

### comm_iface
Returns the hardware interface associated to the XBee device.

**Returns** the hardware interface associated to the XBee device.

**Return type** XBeeCommunicationInterface

**See also:** XBeeSerialPort

### classmethod create_xbee_device (comm_port_data)
Creates and returns an **XBeeDevice** from data of the port to which is connected.

**Parameters**
- **comm_port_data** (*Dictionary*) – dictionary with all comm port data needed.
  - dictionary keys are (The) –
    "baudRate" -> Baud rate.
    "port" -> Port number.
    "bitSize" -> Bit size.
    "stopBits" -> Stop bits.
    "parity" -> Parity.
    "flowControl" -> Flow control.
    "timeout" for -> Timeout for synchronous operations (in seconds).
Returns the XBee device created.

Return type `XBeeDevice`

Raises `SerialException` – if the port you want to open does not exist or is already opened.

See also:

`XBeeDevice`

```python
def del_bluetooth_data_received_callback(callback):
    """Deletes a callback for the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

    Parameters
    CKER: `Function` -- the callback to delete.

    Raises
    CKER: `ValueError` -- if `callback` is not in the callback list of `digi.xbee.reader.BluetoothDataReceived` event.
    """

def del_io_sample_received_callback(callback):
    """Deletes a callback for the callback list of `digi.xbee.reader.IOSampleReceived` event.

    Parameters
    CKER: `Function` -- the callback to delete.

    Raises
    CKER: `ValueError` -- if `callback` is not in the callback list of `digi.xbee.reader.IOSampleReceived` event.
    """

def del_micropython_data_received_callback(callback):
    """Deletes a callback for the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

    Parameters
    CKER: `Function` -- the callback to delete.

    Raises
    CKER: `ValueError` -- if `callback` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.
    """

def del_modem_status_received_callback(callback):
    """Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

    Parameters
    CKER: `Function` -- the callback to delete.

    Raises
    CKER: `ValueError` -- if `callback` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.
    """

def del_packet_received_callback(callback):
    """Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

    Parameters
    CKER: `Function` -- the callback to delete.

    Raises
    CKER: `ValueError` -- if `callback` is not in the callback list of `digi.xbee.reader.PacketReceived` event.
    """

def del_route_received_callback(callback):
    """Deletes a callback for the callback list of `RouteReceived` event.

    Parameters
    CKER: `Function` -- the callback to delete.

    See also:

    `XBeeDevice.add_route_received_callback()`
```
del_socket_data_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.

del_socket_data_received_from_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

del_socket_state_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

del_user_data_relay_received_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.

disable_bluetooth()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

disable_bluetooth()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
**execute_command** *(parameter)*

Executes the provided command.

*Parameters* **parameter** *(String)* – The name of the AT command to be executed.

*Raises*

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**flush_queues** ()

Flushes the packets queue.

**get_64bit_addr** ()

Returns the 64-bit address of the XBee device.

*Returns*  the 64-bit address of the XBee device.

*Return type*  **XBee64BitAddress**

*See also:  **XBee64BitAddress**

**get_adc_value** *(io_line)*

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use **AbstractXBeeDevice. set_io_configuration** () and **IOMode.ADC**.

*Parameters* **io_line** *(IOLine)* – the IO line to get its ADC value.

*Returns*  the analog value corresponding to the provided IO line.

*Return type*  **Integer**

*Raises*

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

*See also:  **IOLine**
get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

Returns the API output mode of the XBee device.

Return type APIOutputMode

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns the parameter value.

Return type Bytearray

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.
Returns  The Bluetooth MAC address.

Return type  String

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

```
get_current_frame_id()
```

Returns the last used frame ID.

- Returns  the last used frame ID.
- Return type  Integer

```
get_dio_value(io_line)
```

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

- Parameters io_line (IOLine) – the DIO line to gets its digital value.
- Returns  current value of the provided IO line.
- Return type  IOValue

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

```
IOLine
IOValue
```

```
get_firmware_version()
```

Returns the firmware version of the XBee device.

- Returns  the hardware version of the XBee device.
- Return type  Bytearray

```
get_hardware_version()
```

Returns the hardware version of the XBee device.

- Returns  the hardware version of the XBee device.
- Return type  HardwareVersion
See also:

*HardwareVersion*

**get_io_configuration**(io_line)

Returns the configuration of the provided IO line.

**Parameters**

- **io_line** (*IOLine*) — the io line to configure.

**Returns**

the IO mode of the IO line provided.

**Return type** *IOMode*

**Raises**

- *TimeoutException* — if the response is not received before the read timeout expires.
- *XBeeException* — if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* — if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* — if the response is not as expected.
- *OperationNotSupportedException* — if the received data is not an IO mode.

**get_io_sampling_rate**()

Returns the IO sampling rate of the XBee device.

**Returns**

the IO sampling rate of XBee device.

**Return type** *Integer*

**Raises**

- *TimeoutException* — if the response is not received before the read timeout expires.
- *XBeeException* — if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* — if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* — if the response is not as expected.

**get_next_frame_id**()

Returns the next frame ID of the XBee device.

**Returns**

The next frame ID of the XBee device.

**Return type** *Integer*

**get_node_id**()

Returns the Node Identifier (*NI*) value of the XBee device.

**Returns**

the Node Identifier (*NI*) of the XBee device.

**Return type** *String*

**get_parameter**(param, parameter_value=None)

Override.

See also:
AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee device.

Returns the power level of the XBee device.

Return type PowerLevel

Raises TimeoutException – if the response is not received before the read timeout expires.

See also:

PowerLevel

get_protocol()
Returns the current protocol of the XBee device.

Returns the current protocol of the XBee device.

Return type XBeeProtocol

See also:

XBeeProtocol

get_pwm_duty_cycle(io_line)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters io_line (IOLine) – the IO line to get its PWM duty cycle.

Returns the PWM duty cycle of the given IO line or None if the response is empty.

Return type Integer

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

• ValueError – if the passed IO_LINE has no PWM capability.

See also:

IOLine

get_role()
Gets the XBee role.

Returns the role of the XBee.
get_route_to_node (remote, timeout=10)

Gets the route from this XBee to the given remote node.

**Parameters**

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.

**Returns**

Tuple containing route data *(None if the route was not read in the provided timeout):*

- source (*RemoteXBeeDevice*) – The source node of the route.
- destination (*RemoteXBeeDevice*) – The destination node of the route.
- hops (*List*) – List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

**Return type** Tuple

get_sync_ops_timeout ()

Returns the serial port read timeout.

**Returns** the serial port read timeout in seconds.

**Return type** Integer

get_xbee_device_callbacks ()

Returns this XBee internal callbacks for process received packets.

This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

**Returns** PacketReceived

has_explicit_packets ()

Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

**Returns** True if this XBee device’s queue has explicit packets, False otherwise.

**Return type** Boolean

See also:

XBeeDevice.has_packets ()

has_packets ()

Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.
Returns True if this XBee device’s queue has packets, False otherwise.

Return type Boolean

See also:

`XBeeDevice.has_explicit_packets()`

`is_apply_changes_enabled()`

Returns whether the apply_changes flag is enabled or not.

Returns True if the apply_changes flag is enabled, False otherwise.

Return type Boolean

`is_open()`

Returns whether this XBee device is open or not.

Returns Boolean. True if this XBee device is open, False otherwise.

`is_remote()`

Override method.

See also:

`AbstractXBeeDevice.is_remote()`

`log`

Returns the XBee device log.

Returns the XBee device logger.

Return type Logger

`open(force_settings=False)`

Opens the communication with the XBee device and loads some information about it.

Parameters `force_settings` (Boolean, optional) – True to open the device ensuring/forcing that the specified serial settings are applied even if the current configuration is different, False to open the device with the current configuration. Default to False.

Raises

- `TimeoutException` – if there is any problem with the communication.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – if the XBe device is already open.

`operating_mode`

Returns this XBee device’s operating mode.

Returns `OperatingMode`. This XBe device’s operating mode.

`reachable`

Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.
Return type Boolean

read_io_sample()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type IOSample

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

IOSample

reset()
Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(data)
Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters data (Bytearray) – Data to send.

Raises
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – if there is any problem sending the data.

See also:

XBeeDevice.send_micropython_data()
XBeeDevice.send_user_data_relay()
send_micropython_data \( (\text{data}) \)
Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters \( \text{data} (\text{Bytearray}) \) – Data to send.

Raises

- \( \text{InvalidOperatingModeException} \) – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \( \text{XBeeException} \) – if there is any problem sending the data.

See also:

\[ \text{XBeeDevice.send_bluetooth_data()} \]
\[ \text{XBeeDevice.send_user_data_relay()} \]

send_packet \( (\text{packet}, \text{sync}=\text{False}) \)
Override method.

See also:

\[ \text{AbstractXBeeDevice._send_packet()} \]

send_packet_sync_and_get_response \( (\text{packet_to_send}, \text{timeout}=\text{None}) \)
Override method.

See also:

\[ \text{AbstractXBeeDevice._send_packet_sync_and_get_response()} \]

send_user_data_relay \( (\text{local_interface}, \text{data}) \)
Sends the given data to the given XBee local interface.

Parameters

- \( \text{local_interface} (\text{XBeeLocalInterface}) \) – Destination XBee local interface.
- \( \text{data} (\text{Bytearray}) \) – Data to send.

Raises

- \( \text{InvalidOperatingModeException} \) – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- \( \text{ValueError} \) – if \( \text{local_interface} \) is None.
- \( \text{XBeeException} \) – if there is any problem sending the User Data Relay.

See also:

\[ \text{XBeeLocalInterface} \]
serial_port

Returns the serial port associated to the XBee device, if any.

Returns

the serial port associated to the XBee device. Returns None if the local XBee does not use serial communication.

Return type: XBeeSerialPort

See also:

XBeeSerialPort

set_16bit_addr(value)

Sets the 16-bit address of the XBee device.

Parameters:

value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises:

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is not 802.15.4.

set_api_output_mode(api_output_mode)

Deprecated since version 1.3: Use set_api_output_mode_value()

Sets the API output mode of the XBee device.

Parameters:

api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

Raises:

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is ZigBee

See also:

APIOutputMode

set_api_output_mode_value(api_output_mode)

Sets the API output mode of the XBee.
Parameters `api_output_mode` *(Integer)* – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

**See also:**

`digi.xbee.models.mode.APIOutputModeBit`

### `set_dio_change_detection` *(io_lines_set)*

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A `None` set of lines disables this feature.

**Parameters** `io_lines_set` – set of `IOLine`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**See also:**

`IOLine`

### `set_dio_value` *(io_line, io_value)*

Sets the digital value (high or low) to the provided IO line.

**Parameters**

- `io_line` *(`IOLine`)* – the digital IO line to sets its value.
- `io_value` *(`IOValue`)* – the IO value to set to the IO line.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.


- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.

See also:

*IOLine*
*IOWValue*

**set_io_configuration**(io_line, io_mode)

Sets the configuration of the provided IO line.

Parameters

- **io_line**(IOLine) – the IO line to configure.
- **io_mode**(IOMode) – the IO mode to set to the IO line.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

*IOLine*
*IOMode*

**set_io_sampling_rate**(rate)

Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate**(Integer) – the new IO sampling rate of the XBee device in seconds.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**set_node_id**(node_id)

Sets the Node Identifier (NI) value of the XBee device.

Parameters **node_id**(String) – the new Node Identifier (NI) of the XBee device.

Raises

- **ValueError** – if node_id is None or its length is greater than 20.
• TimeoutException – if the response is not received before the read timeout expires.

```python
set_parameter(param, value)
```
Override.

See: AbstractXBeeDevice.set_parameter()

```python
set_power_level(power_level)
```
Sets the power level of the XBee device.

**Parameters**

- `power_level (PowerLevel)` – the new power level of the XBee device.

**Raises** TimeoutException – if the response is not received before the read timeout expires.

See also:

PowerLevel

```python
set_pwm_duty_cycle(io_line, cycle)
```
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

**Parameters**

- `io_line (IOLine)` – the IO Line to be assigned.
- `cycle (Integer)` – duty cycle in % to be assigned. Must be between 0 and 100.

**Raises**

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- ValueError – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

IOLine

IOMode.PWM

```python
set_sync_ops_timeout(sync_ops_timeout)
```
Sets the serial port read timeout.

**Parameters**

- `sync_ops_timeout (Integer)` – the read timeout, expressed in seconds.

```python
update_bluetooth_password(new_password)
```
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**

- `new_password (String)` – New Bluetooth password.

**Raises**
TimeoutException – if the response is not received before the read timeout expires.

XBeeException – if the XBee device’s serial port is closed.

InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

update_device_data_from(device)

Updates the current device reference with the data provided for the given device.

This is only for internal use.

Parameters device(AbstractXBeeDevice) – the XBee device to get the data from.

Returns True if the device data has been updated, False otherwise.

Return type Boolean

update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)

Performs a firmware update operation of the device.

Parameters

• xml_firmware_file(String) – path of the XML file that describes the firmware to upload.

• xbee_firmware_file(String, optional) – location of the XBee binary firmware file.

• bootloader_firmware_file(String, optional) – location of the bootloader binary firmware file.

• timeout(Integer, optional) – the maximum time to wait for target read operations during the update process.

• progress_callback(Function, optional) – function to execute to receive progress information. Receives two arguments:

  – The current update task as a String
  – The current update task percentage as an Integer

Raises

• XBeeException – if the device is not open.

• InvalidOperatingModeException – if the device operating mode is invalid.

• OperationNotSupportedException – if the firmware update is not supported in the XBee device.

• FirmwareUpdateException – if there is any error performing the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use
method `is_apply_configuration_changes_enabled()` to get its status and `enable_apply_configuration_changes()` to enable/disable the option. If it is disabled, method `apply_changes()` can be used in order to manually apply the changes.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

class `digi.xbee.devices.CellularDevice` (port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)

Bases: `digi.xbee.devices.IPDevice`

This class represents a local Cellular device.

Class constructor. Instantiates a new `CellularDevice` with the provided parameters.

Parameters

- `port` (Integer or String) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- `baud_rate` (Integer) – the serial port baud rate.
- (Integer, default (_sync_ops_timeout) – serial.EIGHTBITS): comm port bitsize.
- (Integer, default – serial.STOPBITS_ONE): comm port stop bits.
- (Character, default (parity) – serial.PARITY_NONE): comm port parity.
- (Integer, default – FlowControl.NONE): comm port flow control.
- (Integer, default – 3): the read timeout (in seconds).
- `comm_iface` (XBeeCommunicationInterface) – the communication interface.

:raises All exceptions raised by `XBeeDevice.__init__()` constructor.: See also:

`XBeeDevice`

`XBeeDevice.__init__()`

`open` (force_settings=False)

Override.

Raises

- `TimeoutException` – If there is any problem with the communication.
• `InvalidOperatingModeException` – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `XBeeException` – If the protocol is invalid or if the XBee device is already open.

See also:

```
XBeeDevice.open()
```

```
get_protocol()
```

Override.

See also:

```
XBeeDevice.get_protocol()
```

```
read_device_info(init=True, fire_event=True)
```

Override.

See also:

```
XBeeDevice.read_device_info()
```

```
is_connected()
```

Returns whether the device is connected to the Internet or not.

Returns `True` if the device is connected to the Internet, `False` otherwise.

Return type `Boolean`

Raises

• `TimeoutException` – if there is a timeout getting the association indication status.

• `XBeeException` – if there is any other XBee related exception.

```
get_cellular_ai_status()
```

Returns the current association status of this Cellular device.

It indicates occurrences of errors during the modem initialization and connection.

Returns The association indication status of the Cellular device.

Return type `CellularAssociationIndicationStatus`

Raises

• `TimeoutException` – if there is a timeout getting the association indication status.

• `XBeeException` – if there is any other XBee related exception.

```
add_sms_callback(callback)
```

Adds a callback for the event `digi.xbee.reader.SMSReceived`.

Parameters `callback` (`Function`) – the callback. Receives one argument.

• The data received as an `digi.xbee.models.message.SMSMessage`
del_sms_callback(callback)
Deletes a callback for the callback list of `digi.xbee.reader.SMSReceived` event.

**Parameters**
`callback Function` – the callback to delete.

**Raises**
`ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SMSReceived` event.

get_imei_addr()
Returns the IMEI address of this Cellular device.

To refresh this value use the method `CellularDevice.read_device_info()`.

**Returns**
The IMEI address of this Cellular device.

**Return type**
XBeeIMEIAddress

send_sms(phone_number, data)
Sends the provided SMS message to the given phone number.

This method blocks till a success or error response arrives or the configured receive timeout expires.

For non-blocking operations use the method `CellularDevice.send_sms_async()`.

**Parameters**
- `phone_number String` – The phone number to send the SMS to.
- `data String` – Text of the SMS.

**Raises**
- `ValueError` – if `phone_number` is None.
- `ValueError` – if `data` is None.
- `OperationNotSupportedException` – if the device is remote.
- `TimeoutException` – if there is a timeout sending the SMS.
- `XBeeException` – if there is any other XBee related exception.

send_sms_async(phone_number, data)
Sends asynchronously the provided SMS to the given phone number.

Asynchronous transmissions do not wait for answer or for transmit status packet.

**Parameters**
- `phone_number String` – The phone number to send the SMS to.
- `data String` – Text of the SMS.

**Raises**
- `ValueError` – if `phone_number` is None.
- `ValueError` – if `data` is None.
- `OperationNotSupportedException` – if the device is remote.
- `XBeeException` – if there is any other XBee related exception.

get_sockets_list()
Returns a list with the IDs of all active (open) sockets.

**Returns**
list with the IDs of all active (open) sockets, or empty list if there is not any active socket.
Return type  List

Raises

- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.

`get_socket_info(socket_id)`
Returns the information of the socket with the given socket ID.

Parameters  
`socket_id` (`Integer`) – ID of the socket.

Returns  
The socket information, or `None` if the socket with that ID does not exist.

Return type  `SocketInfo`

Raises

- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.

See also:

`SocketInfo`

`get_64bit_addr()`  
Deprecated.

Cellular protocol does not have an associated 64-bit address.

`add_io_sample_received_callback(callback)`  
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

`del_io_sample_received_callback(callback)`  
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

`set_dio_change_detection(io_lines_set)`  
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

`get_io_sampling_rate()`  
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

`set_io_sampling_rate(rate)`  
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.
get_node_id()
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

set_node_id(node_id)
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

get_power_level()
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

set_power_level(power_level)
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

add_bluetooth_data_received_callback (callback)
    Adds a callback for the event digi.xbee.reader.BluetoothDataReceived.
    Parameters callback (Function) – the callback. Receives one argument.
    • The Bluetooth data as a Bytearray

add_data_received_callback (callback)
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

add_expl_data_received_callback (callback)
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

add_ip_data_received_callback (callback)
    Adds a callback for the event digi.xbee.reader.IPDataReceived.
    Parameters callback (Function) – the callback. Receives one argument.
    • The data received as an digi.xbee.models.message.IPMessage

add_micropython_data_received_callback (callback)
    Adds a callback for the event digi.xbee.reader.MicroPythonDataReceived.
    Parameters callback (Function) – the callback. Receives one argument.
    • The MicroPython data as a Bytearray

add_modem_status_received_callback (callback)
    Adds a callback for the event digi.xbee.reader.ModemStatusReceived.
    Parameters callback (Function) – the callback. Receives one argument.
    • The modem status as a digi.xbee.models.status.ModemStatus

add_packet_received_callback (callback)
    Adds a callback for the event digi.xbee.reader.PacketReceived.
    Parameters callback (Function) – the callback. Receives two arguments.
    • The received packet as a digi.xbee.packets.base.XBeeAPIPacket

add_route_received_callback (callback)
    Adds a callback for the event RouteReceived. This works for Zigbee and Digimesh devices.
Parameters `callback (Function)` – the callback. Receives three arguments.

- `source (XBeeDevice)`: The source node.
- `destination (RemoteXBeeDevice)`: The destination node.
- `hops (List)`: List of intermediate hops from closest to source to closest to destination (`RemoteXBeeDevice`).

See also:

```python
XBeeDevice.del_route_received_callback()
```

`add_socket_data_received_callback (callback)`

Adds a callback for the event `digi.xbee.reader.SocketDataReceived`.

Parameters `callback (Function)` – the callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray

`add_socket_data_received_from_callback (callback)`

Adds a callback for the event `digi.xbee.reader.SocketDataReceivedFrom`.

Parameters `callback (Function)` – the callback. Receives three arguments.

- The socket ID as an Integer.
- A pair (host, port) of the source address where host is a string representing an IPv4 address like ’100.50.200.5’, and port is an integer.
- The data received as Bytearray

`add_socket_state_received_callback (callback)`

Adds a callback for the event `digi.xbee.reader.SocketStateReceived`.

Parameters `callback (Function)` – the callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a `SocketState`

`add_user_data_relay_received_callback (callback)`

Adds a callback for the event `digi.xbee.reader.RelayDataReceived`.

Parameters `callback (Function)` – the callback. Receives one argument.

- The relay data as a `digi.xbee.models.message.UserDataRelayMessage`

`apply_changes ()`

Applies changes via AC command.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
apply_profile(profile_path, progress_callback=None)
Applies the given XBee profile to the XBee device.

Parameters

- **profile_path** *(String)* – path of the XBee profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **UpdateProfileException** – if there is any error applying the XBee profile.
- **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

close()
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

comm_iface
Returns the hardware interface associated to the XBee device.

Returns the hardware interface associated to the XBee device.

Return type XBeeCommunicationInterface

See also:

XBeeSerialPort

classmethod create_xbee_device(comm_port_data)
Creates and returns an XBeeDevice from data of the port to which it is connected.

Parameters

- **comm_port_data** *(Dictionary)* – dictionary with all comm port data needed.

  * dictionary keys are (The) –

    - “baudRate” – Baud rate.
    - “port” – Port number.
    - “bitSize” – Bit size.
    - “stopBits” – Stop bits.
    - “parity” – Parity.
    - “flowControl” – Flow control.
    - “timeout” for – Timeout for synchronous operations (in seconds).

Returns the XBee device created.
Return type **XBeeDevice**

Raises **SerialException** – if the port you want to open does not exist or is already opened.

See also:

**XBeeDevice**

### del_bluetooth_data_received_callback

```python
(del_bluetooth_data_received_callback)(callback)
```

Deletes a callback for the callback list of **digi.xbee.reader.BluetoothDataReceived** event.

**Parameters**

- `callback`: (Function) – the callback to delete.

** Raises**

- **ValueError** – if `callback` is not in the callback list of **digi.xbee.reader.BluetoothDataReceived** event.

### del_data_received_callback

```python
(del_data_received_callback)(callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an **AttributeError**.

### del_expl_data_received_callback

```python
(del_expl_data_received_callback)(callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an **AttributeError**.

### del_ip_data_received_callback

```python
(del_ip_data_received_callback)(callback)
```

Deletes a callback for the callback list of **digi.xbee.reader.IPDataReceived** event.

**Parameters**

- `callback`: (Function) – the callback to delete.

** Raises**

- **ValueError** – if `callback` is not in the callback list of **digi.xbee.reader.IPDataReceived** event.

### del_micropython_data_received_callback

```python
(del_micropython_data_received_callback)(callback)
```

Deletes a callback for the callback list of **digi.xbee.reader.MicroPythonDataReceived** event.

**Parameters**

- `callback`: (Function) – the callback to delete.

** Raises**

- **ValueError** – if `callback` is not in the callback list of **digi.xbee.reader.MicroPythonDataReceived** event.

### del_modem_status_received_callback

```python
(del_modem_status_received_callback)(callback)
```

Deletes a callback for the callback list of **digi.xbee.reader.ModemStatusReceived** event.

**Parameters**

- `callback`: (Function) – the callback to delete.

** Raises**

- **ValueError** – if `callback` is not in the callback list of **digi.xbee.reader.ModemStatusReceived** event.

### del_packet_received_callback

```python
(del_packet_received_callback)(callback)
```

Deletes a callback for the callback list of **digi.xbee.reader.PacketReceived** event.

**Parameters**

- `callback`: (Function) – the callback to delete.

** Raises**

- **ValueError** – if `callback` is not in the callback list of **digi.xbee.reader.PacketReceived** event.

### del_route_received_callback

```python
(del_route_received_callback)(callback)
```

Deletes a callback for the callback list of **RouteReceived** event.

**Parameters**

- `callback`: (Function) – the callback to delete.
See also:

XBeeDevice.add_route_received_callback()

**del_socket_data_received_callback** *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

- **Parameters** callback *(Function)* – the callback to delete.
- **Raises** `ValueError` – if callback is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.

**del_socket_data_received_from_callback** *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

- **Parameters** callback *(Function)* – the callback to delete.
- **Raises** `ValueError` – if callback is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

**del_socket_state_received_callback** *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

- **Parameters** callback *(Function)* – the callback to delete.
- **Raises** `ValueError` – if callback is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

**del_user_data_relay_received_callback** *(callback)*
Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

- **Parameters** callback *(Function)* – the callback to delete.
- **Raises** `ValueError` – if callback is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.

**disable_bluetooth** *
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**enable_apply_changes** *(value)*
Sets the apply changes flag.

- **Parameters** value *(Boolean)* – True to enable the apply changes flag, False to disable it.

**enable_bluetooth** *
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.
Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

`execute_command`(*parameter*)

Executes the provided command.

Parameters `parameter` (*String*) – The name of the AT command to be executed.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

`flush_queues`()

Flushes the packets queue.

`get_16bit_addr`()

Deprecated.

This protocol does not have an associated 16-bit address.

`get_adc_value`(*io_line*)

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use `AbstractXBeeDevice.set_io_configuration()` and `IOMode.ADC`.

Parameters `io_line` (*IOLine*) – the IO line to get its ADC value.

Returns the analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

`IOLine`
get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

Returns  the API output mode of the XBee device.

Return type  APIOutputMode

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns  the parameter value.

Return type  Bytearray

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.
Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

`get_current_frame_id()`

Returns the last used frame ID.

Returns the last used frame ID.

Return type Integer

`get_dest_address()`

Deprecated.

Operation not supported in this protocol. Use `IPDevice.get_dest_ip_addr()` instead. This method will raise an `AttributeError`.

`get_dest_ip_addr()`

Returns the destination IP address.

Returns The configured destination IP address.

Return type `ipaddress.IPv4Address`

Raises

- `TimeoutException` – if there is a timeout getting the destination IP address.
- `XBeeException` – if there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

`get_dio_value(io_line)`

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

Parameters `io_line` (IOLine) – the DIO line to gets its digital value.

Returns current value of the provided IO line.

Return type `IOValue`

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
• **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

See also:

*IOLine*

*IOValue*

get**_firmware_version_**()

Returns the firmware version of the XBee device.

**Returns** the hardware version of the XBee device.

**Return type** Bytearray

get**_hardware_version_**()

Returns the hardware version of the XBee device.

**Returns** the hardware version of the XBee device.

**Return type** HardwareVersion

See also:

*HardwareVersion*

get**_io_configuration_(io_line)_**

Returns the configuration of the provided IO line.

**Parameters** *io_line* (*IOLine*) – the io line to configure.

**Returns** the IO mode of the IO line provided.

**Return type** *IOMode*

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the received data is not an IO mode.

get**_ip_addr_**()

Returns the IP address of this IP device.

To refresh this value use the method *IPDevice.read_device_info()*.

**Returns** The IP address of this IP device.

**Return type** *ipaddress.IPv4Address*

See also:
get_network()
Deprecated.
This protocol does not support the network functionality.

get_next_frame_id()
Returns the next frame ID of the XBee device.

Returns  The next frame ID of the XBee device.

Return type  Integer

get_pan_id()
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

get_parameter(param, parameter_value=None)
Override.
See also:

AbstractXBeeDevice.get_parameter()

get_pwm_duty_cycle(io_line)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters  io_line (IOLine) – the IO line to get its PWM duty cycle.

Returns  the PWM duty cycle of the given IO line or None if the response is empty.

Return type  Integer

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBe device’s serial port is closed.
- InvalidOperatingModeException – if the XBe device’s operating mode is not
API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- ValueError – if the passed IO_LINE has no PWM capability.

See also:

IOLine

get_role()
Gets the XBee role.

Returns  the role of the XBee.

Return type  digi.xbee.models.protocol.Role
get_route_to_node (remote, timeout=10)

Gets the route from this XBee to the given remote node.

Parameters

• remote (RemoteXBeeDevice) – The remote node.
• timeout (Float, optional, default=10) – Maximum number of seconds to wait for the route.

Returns

Tuple containing route data (None if the route was not read in the provided timeout):
- source (RemoteXBeeDevice): The source node of the route.
- destination (RemoteXBeeDevice): The destination node of the route.
- hops (List): List of intermediate nodes (RemoteXBeeDevice) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout ()

Returns the serial port read timeout.

Returns the serial port read timeout in seconds.

Return type Integer

get_xbee_device_callbacks ()

Returns this XBee internal callbacks for process received packets.

This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

Returns PacketReceived

has_explicit_packets ()

Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

Returns True if this XBee device’s queue has explicit packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets ()

has_packets ()

Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

Returns True if this XBee device’s queue has packets, False otherwise.
Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()
Returns whether the apply_changes flag is enabled or not.

Returns True if the apply_changes flag is enabled, False otherwise.

Return type Boolean

is_open()
Returns whether this XBee device is open or not.

Returns Boolean. True if this XBee device is open, False otherwise.

is_remote()
Override method.

See also:

AbstractXBeeDevice.is_remote()

log
Returns the XBee device log.

Returns the XBee device logger.

Return type Logger

operating_mode
Returns this XBee device’s operating mode.

Returns OperatingMode. This XBee device’s operating mode.

reachable
Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

read_data(timeout=None, explicit=False)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

read_data_from(remote_xbee_device, timeout=None, explicit=False)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

read_io_sample()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.
Return type **IOSample**

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

**IOSample**

**read_ip_data** *(timeout=3)*

Reads new IP data received by this XBee device during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

**Parameters**

- **timeout** *(Integer, optional)* – The time to wait for new IP data in seconds.

**Returns**

IP message, *None* if this device did not receive new data.

**Return type** **IPMessage**

**Raises** **ValueError** – if timeout is less than 0.

**read_ip_data_from** *(ip_addr, timeout=3)*

Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

**Parameters**

- **ip_addr** *(ipaddress.IPv4Address)* – The IP address to read data from.
- **timeout** *(Integer, optional)* – The time to wait for new IP data in seconds.

**Returns**

IP message, *None* if this device did not receive new data from the provided IP address.

**Return type** **IPMessage**

**Raises** **ValueError** – if timeout is less than 0.
reset()
    Override method.
    See also:

    AbstractXBeeDevice.reset()

scan_counter
    Returns the scan counter for this node.
    Returns  The scan counter for this node.
    Return type  Integer

send_bluetooth_data(data)
    Sends the given data to the Bluetooth interface using a User Data Relay frame.
    Parameters  data (Bytearray) – Data to send.
    Raises
    • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
    • XBeeException – if there is any problem sending the data.
    See also:

    XBeeDevice.send_micropython_data()
    XBeeDevice.send_user_data_relay()

send_data (remote_xbee_device, data, transmit_options=0)
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

send_data_async (remote_xbee_device, data, transmit_options=0)
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

send_data_broadcast (data, transmit_options=0)
    Deprecated.
    Operation not supported in this protocol. This method will raise an AttributeError.

send_ip_data (ip_addr, dest_port, protocol, data, close_socket=False)
    Sends the provided IP data to the given IP address and port using the specified IP protocol. For TCP and TCP SSL protocols, you can also indicate if the socket should be closed when data is sent.
    This method blocks till a success or error response arrives or the configured receive timeout expires.
    Parameters
    • ip_addr (ipaddress.IPv4Address) – The IP address to send IP data to.
    • dest_port (Integer) – The destination port of the transmission.
    • protocol (IPProtocol) – The IP protocol used for the transmission.
• **data** *(String or Bytearray)* – The IP data to be sent.

• **close_socket** *(Boolean, optional)* – True to close the socket just after the transmission. False to keep it open. Default to False.

**Raises**

• **ValueError** – if ip_addr is None.

• **ValueError** – if protocol is None.

• **ValueError** – if data is None.

• **ValueError** – if dest_port is less than 0 or greater than 65535.

• **OperationNotSupportedException** – if the device is remote.

• **TimeoutException** – if there is a timeout sending the data.

• **XBeeException** – if there is any other XBee related exception.

**send_ip_data_async(ip_addr, dest_port, protocol, data, close_socket=False)**

Sends the provided IP data to the given IP address and port asynchronously using the specified IP protocol. For TCP and TCP SSL protocols, you can also indicate if the socket should be closed when data is sent.

Asynchronous transmissions do not wait for answer from the remote device or for transmit status packet.

**Parameters**

• **ip_addr** *(ipaddress.IPv4Address)* – The IP address to send IP data to.

• **dest_port** *(Integer)* – The destination port of the transmission.

• **protocol** *(IPProtocol)* – The IP protocol used for the transmission.

• **data** *(String or Bytearray)* – The IP data to be sent.

• **close_socket** *(Boolean, optional)* – True to close the socket just after the transmission. False to keep it open. Default to False.

**Raises**

• **ValueError** – if ip_addr is None.

• **ValueError** – if protocol is None.

• **ValueError** – if data is None.

• **ValueError** – if dest_port is less than 0 or greater than 65535.

• **OperationNotSupportedException** – if the device is remote.

• **XBeeException** – if there is any other XBee related exception.

**send_ip_data_broadcast(dest_port, data)**

Sends the provided IP data to all clients.

This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

**Parameters**

• **dest_port** *(Integer)* – The destination port of the transmission.

• **data** *(String or Bytearray)* – The IP data to be sent.

**Raises**

• **ValueError** – if data is None.

• **ValueError** – if dest_port is less than 0 or greater than 65535.
• `TimeoutException` – if there is a timeout sending the data.
• `XBeeException` – if there is any other XBee related exception.

`send_micropython_data(data)`
Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters `data` (*Bytearray*) – Data to send.

Raises
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `XBeeException` – if there is any problem sending the data.

See also:

```
XBeeDevice.send_bluetooth_data()
XBeeDevice.send_user_data_relay()
```

`send_packet(packet, sync=False)`
Override method.

See also:

```
AbstractXBeeDevice._send_packet()
```

`send_packet_sync_and_get_response(packet_to_send, timeout=None)`
Override method.

See also:

```
AbstractXBeeDevice._send_packet_sync_and_get_response()
```

`send_user_data_relay(local_interface, data)`
Sends the given data to the given XBee local interface.

Parameters
• `local_interface` (*XBeeLocalInterface*) – Destination XBee local interface.
• `data` (*Bytearray*) – Data to send.

Raises
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ValueError` – if `local_interface` is `None`.
• `XBeeException` – if there is any problem sending the User Data Relay.

See also:
XBeeLocalInterface

serial_port
Returns the serial port associated to the XBee device, if any.

Returns
the serial port associated to the XBee device. Returns None if the local XBee does not use serial communication.

Return type
XBeeSerialPort

See also:

XBeeSerialPort

set_16bit_addr(value)
Sets the 16-bit address of the XBee device.

Parameters
value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is not 802.15.4.

set_api_output_mode(api_output_mode)
Deprecated since version 1.3: Use set_api_output_mode_value()

Sets the API output mode of the XBee device.

Parameters
api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is ZigBee

See also:

APIOutputMode
**set_api_output_mode_value**(api_output_mode)

Sets the API output mode of the XBee.

**Parameters**

api_output_mode (**Integer**) – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

**set_dest_address**(addr)

Deprecated.

Operation not supported in this protocol. Use `IPDevice.set_dest_ip_addr()` instead. This method will raise an `AttributeError`.

**set_dest_ip_addr**(address)

Sets the destination IP address.

**Parameters**

- address (**ipaddress.IPv4Address**) – Destination IP address.

**Raises**

- `ValueError` – if address is `None`.
- `TimeoutException` – if there is a timeout setting the destination IP address.
- `XBeeException` – if there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

**set_dio_value**(io_line, io_value)

Sets the digital value (high or low) to the provided IO line.

**Parameters**

- io_line (**IOLine**) – the digital IO line to set its value.
- io_value (**IOValue**) – the IO value to set to the IO line.

**Raises**
set_io_configuration \((io\_line, io\_mode)\)

Sets the configuration of the provided IO line.

Parameters

- **io_line** \((IOLine)\) – the IO line to configure.
- **io_mode** \((IOMode)\) – the IO mode to set to the IO line.

Raise

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

IOLine
IOMode

set_pan_id \((value)\)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

set_parameter \((param, value)\)

Override.

See: AbstractXBeeDevice.set_parameter()

set_pwm_duty_cycle \((io\_line, cycle)\)

Sets the duty cycle in \% of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

- **io_line** \((IOLine)\) – the IO Line to be assigned.
- **cycle** \((Integer)\) – duty cycle in \% to be assigned. Must be between 0 and 100.

Raise

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

IOLine
IOMode
• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBees device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **ValueError** – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

*IOLine*

*IOMode.PWM*

**set_sync_ops_timeout** *(sync_ops_timeout)*

Sets the serial port read timeout.

**Parameters**

* sync_ops_timeout *(Integer)* – the read timeout, expressed in seconds.

**start_listening** *(source_port)*

Starts listening for incoming IP transmissions in the provided port.

**Parameters**

* source_port *(Integer)* – Port to listen for incoming transmissions.

**Raises**

• **ValueError** – if source_port is less than 0 or greater than 65535.

• **TimeoutException** – if there is a timeout setting the source port.

• **XBeeException** – if there is any other XBee related exception.

**stop_listening** *

Stops listening for incoming IP transmissions.

**Raises**

• **TimeoutException** – if there is a timeout processing the operation.

• **XBeeException** – if there is any other XBee related exception.

**update_bluetooth_password** *(new_password)*

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**

* new_password *(String)* – New Bluetooth password.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBees device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from** *(device)*

Updates the current device reference with the data provided for the given device.

This is only for internal use.
Parameters  

Parameters **device** (*AbstractXBeeDevice*) – the XBee device to get the data from.

Returns  

Returns **True** if the device data has been updated, **False** otherwise.

Return type  

Return type **Boolean**

**update_firmware** (*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, 
*timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the device.

Parameters

- **xml_firmware_file** (*String*) – path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional*) – location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional*) – location of the bootloader binary firmware file.
- **timeout** (*Integer, optional*) – the maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.
- **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes** ()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method **is_apply_configuration_changes_enabled()** to get its status and **enable_apply_configuration_changes()** to enable/disable the option. If it is disabled, method **apply_changes()** can be used in order to manually apply the changes.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

class digi.xbee.devices.LPWANDevice

    (port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)

Bases: digi.xbee.devices.CellularDevice

This class provides common functionality for XBee Low-Power Wide-Area Network devices.

Class constructor. Instantiates a new LPWANDevice with the provided parameters.

Parameters

• **port** (Integer or String) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.

• **baud_rate** (Integer) – the serial port baud rate.

• **(Integer, default (_sync_ops_timeout) = serial.EIGHTBITS):** comm port bitsize.

• **(Integer, default – serial.STOPBITS_ONE):** comm port stop bits.

• **(Character, default (parity) = serial.PARITY_NONE):** comm port parity.

• **(Integer, default – FlowControl.NONE):** comm port flow control.

• **(Integer, default – 3):** the read timeout (in seconds).

• **comm_iface**(XBeeCommunicationInterface) – the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor:

See also:

CellularDevice

CellularDevice.__init__()

**send_ip_data** *(ip_addr, dest_port, protocol, data, close_socket=False)*

Sends the provided IP data to the given IP address and port using the specified IP protocol.

This method blocks till a success or error response arrives or the configured receive timeout expires.

Parameters

• **ip_addr** *(ipaddress.IPv4Address) – The IP address to send IP data to.*

• **dest_port** (Integer) – The destination port of the transmission.

• **protocol** *(IPProtocol) – The IP protocol used for the transmission.*

• **data** *(String or Bytearray) – The IP data to be sent.*

• **close_socket** *(Boolean, optional) – Must be False.*

**Raises** ValueError – if protocol is not UDP.
send_ip_data_async(ip_addr, dest_port, protocol, data, close_socket=False)

Sends the provided IP data to the given IP address and port asynchronously using the specified IP protocol.

Asynchronous transmissions do not wait for answer from the remote device or for transmit status packet.

**Parameters**

- `ip_addr` (ipaddress.IPv4Address) – The IP address to send IP data to.
- `dest_port` (Integer) – The destination port of the transmission.
- `protocol` (IPProtocol) – The IP protocol used for the transmission.
- `data` (String or Bytearray) – The IP data to be sent.
- `close_socket` (Boolean, optional) – Must be False.

**Raises** ValueError – if protocol is not UDP.

add_sms_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

del_sms_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

send_sms(phone_number, data)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

send_sms_async(phone_number, data)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

add_bluetooth_data_received_callback(callback)

Adds a callback for the event `digi.xbee.reader.BluetoothDataReceived`.

**Parameters**

- `callback` (Function) – the callback. Receives one argument.

  - The Bluetooth data as a Bytearray

add_data_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

add_expl_data_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

add_io_sample_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

add_ip_data_received_callback(callback)

Adds a callback for the event `digi.xbee.reader.IPDataReceived`.

**Parameters**

- `callback` (Function) – the callback. Receives one argument.

  - The data received as an `digi.xbee.models.message.IPMessage`
add_micropython_data_received_callback(callback)
Adds a callback for the event `digi.xbee.reader.MicroPythonDataReceived`.

**Parameters** callback (Function) – the callback. Receives one argument.
- The MicroPython data as a Bytearray

add_modem_status_received_callback(callback)
Adds a callback for the event `digi.xbee.reader.ModemStatusReceived`.

**Parameters** callback (Function) – the callback. Receives one argument.
- The modem status as a `digi.xbee.models.status.ModemStatus`

add_packet_received_callback(callback)
Adds a callback for the event `digi.xbee.reader.PacketReceived`.

**Parameters** callback (Function) – the callback. Receives two arguments.
- The received packet as a `digi.xbee.packets.base.XBeeAPIPacket`

add_route_received_callback(callback)
Adds a callback for the event `RouteReceived`. This works for Zigbee and Digimesh devices.

**Parameters** callback (Function) – the callback. Receives three arguments.
- source (`XBeeDevice`): The source node.
- destination (`RemoteXBeeDevice`): The destination node.
- hops (List): List of intermediate hops from closest to source to closest to destination (`RemoteXBeeDevice`).

See also:

`XBeeDevice.del_route_received_callback()`

add_socket_data_received_callback(callback)
Adds a callback for the event `digi.xbee.reader.SocketDataReceived`.

**Parameters** callback (Function) – the callback. Receives two arguments.
- The socket ID as an Integer.
- The data received as Bytearray

add_socket_data_received_from_callback(callback)
Adds a callback for the event `digi.xbee.reader.SocketDataReceivedFrom`.

**Parameters** callback (Function) – the callback. Receives three arguments.
- The socket ID as an Integer.
- A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
- The data received as Bytearray

add_socket_state_received_callback(callback)
Adds a callback for the event `digi.xbee.reader.SocketStateReceived`.

**Parameters** callback (Function) – the callback. Receives two arguments.
- The socket ID as an Integer.
• The state received as a `SocketState`

**add_user_data_relay_received_callback** *(callback)*

Adds a callback for the event `digi.xbee.reader.RelayDataReceived`.

**Parameters**

- **callback** *(Function)* – the callback. Receives one argument.

• The relay data as a `digi.xbee.models.message.UserDataRelayMessage`

**apply_changes** ()

Applies changes via `AC` command.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**apply_profile** *(profile_path, progress_callback=None)*

Applies the given XBee profile to the XBee device.

**Parameters**

- **profile_path** *(String)* – path of the XBee profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

**Raises**

- `XBeeException` – if the device is not open.
- `InvalidOperatingModeException` – if the device operating mode is invalid.
- `UpdateProfileException` – if there is any error applying the XBee profile.
- `OperationNotSupportedException` – if XBee profiles are not supported in the XBee device.

**close** ()

Closes the communication with the XBee device.

This method guarantees that all threads running are stopped and the serial port is closed.

**comm_iface**

Returns the hardware interface associated to the XBee device.

**Returns**

the hardware interface associated to the XBee device.

**Return type** `XBeeCommunicationInterface`

**See also:**

`XBeeSerialPort`
classmethod create_xbee_device(comm_port_data)

Creates and returns an XBeeDevice from data of the port to which is connected.

Parameters

- **comm_port_data** (Dictionary) – dictionary with all comm port data needed.
- **dictionary keys are** (The)

  - "baudRate" –> Baud rate.
  - "port" –> Port number.
  - "bitSize" –> Bit size.
  - "stopBits" –> Stop bits.
  - "parity" –> Parity.
  - "flowControl" –> Flow control.
  - "timeout" for –> Timeout for synchronous operations (in seconds).

Returns the XBee device created.

Return type XBeeDevice

Raises SerialException – if the port you want to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(callback)

Deletes a callback for the callback list of digi.xbee.reader.BluetoothDataReceived event.

Parameters **callback** (Function) – the callback to delete.

Raises ValueError – if **callback** is not in the callback list of digi.xbee.reader.BluetoothDataReceived event.

del_data_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

del_expl_data_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

del_io_sample_received_callback(callback)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

del_ip_data_received_callback(callback)

Deletes a callback for the callback list of digi.xbee.reader.IPDataReceived event.

Parameters **callback** (Function) – the callback to delete.

Raises ValueError – if **callback** is not in the callback list of digi.xbee.reader.IPDataReceived event.

del_micropython_data_received_callback(callback)

Deletes a callback for the callback list of digi.xbee.reader.MicroPythonDataReceived event.
Parameters `callback (Function)` – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

**del_modem_status_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

Parameters `callback (Function)` – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

**del_packet_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

Parameters `callback (Function)` – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.PacketReceived` event.

**del_route_received_callback** *(callback)*

Deletes a callback for the callback list of `RouteReceived` event.

Parameters `callback (Function)` – the callback to delete.

See also:

`XBeeDevice.add_route_received_callback()`

**del_socket_data_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

Parameters `callback (Function)` – the callback to delete.

Raise `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.

**del_socket_data_received_from_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

Parameters `callback (Function)` – the callback to delete.

Raise `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

**del_socket_state_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

Parameters `callback (Function)` – the callback to delete.

Raise `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

**del_user_data_relay_received_callback** *(callback)*

Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

Parameters `callback (Function)` – the callback to delete.

Raise `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.
disable_bluetooth()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

disable_bluetooth()

enable_apply_changes(value)
Sets the apply_changes flag.

Parameters value (Boolean) – True to enable the apply changes flag, False to disable it.

enable_bluetooth()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the AbstractXBeeDevice.update_bluetooth_password() method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

execute_command(parameter)
Executes the provided command.

Parameters parameter (String) – The name of the AT command to be executed.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

flush_queues()
Flushes the packets queue.

get_16bit_addr()
Deprecated.

This protocol does not have an associated 16-bit address.

get_64bit_addr()
Deprecated.

Cellular protocol does not have an associated 64-bit address.
get_adc_value(io_line)

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.set_io_configuration() and IOMode.ADC.

Parameters io_line (IOLine) – the IO line to get its ADC value.

Returns the analog value corresponding to the provided IO line.

Return type Integer

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the response does not contain the value for the given IO line.

See also:

IOLine

get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

Returns the API output mode of the XBee device.

Return type APIOutputMode

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.
The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns the parameter value.

Return type: Bytearray

Raises:

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

Returns: The Bluetooth MAC address.

Return type: String

Raises:

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

get_cellular_ai_status()

Returns the current association status of this Cellular device.

It indicates occurrences of errors during the modem initialization and connection.

Returns: The association indication status of the Cellular device.

Return type: CellularAssociationIndicationStatus

Raises:

- `TimeoutException` – if there is a timeout getting the association indication status.
- `XBeeException` – if there is any other XBe related exception.

get_current_frame_id()

Returns the last used frame ID.

Returns: the last used frame ID.

Return type: Integer
get_dest_address()
Deprecated.

Operation not supported in this protocol. Use IDevice.get_dest_ip_addr() instead. This method will raise an AttributeError.

get_dest_ip_addr()
Returns the destination IP address.

Returns The configured destination IP address.

Return type ipaddress.IPv4Address

Raises
  •  TimeoutException – if there is a timeout getting the destination IP address.
  •  XBeeException – if there is any other XBee related exception.

See also:

ipaddress.IPv4Address

get_dio_value(io_line)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use AbstractXBeeDevice.set_io_configuration().

Parameters io_line (IOLine) – the DIO line to gets its digital value.

Returns current value of the provided IO line.

Return type IOValue

Raises
  •  TimeoutException – if the response is not received before the read timeout expires.
  •  XBeeException – if the XBe device’s serial port is closed.
  •  InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  •  ATCommandException – if the response is not as expected.
  •  OperationNotSupportedException – if the response does not contain the value for the given IO line.

See also:

IOLine
IOValue

get_firmware_version()
Returns the firmware version of the XBe device.

Returns the hardware version of the XBe device.

Return type Bytearray
get_hardware_version()  
Returns the hardware version of the XBee device.

    Returns  the hardware version of the XBee device.

    Return type  HardwareVersion

See also:

HardwareVersion

get_imei_addr()  
Returns the IMEI address of this Cellular device.

To refresh this value use the method CellularDevice.read_device_info().

    Returns  The IMEI address of this Cellular device.

    Return type  XBeeIMEIAddress

get_io_configuration(io_line)  
Returns the configuration of the provided IO line.

    Parameters  io_line (IOLine) – the io line to configure.

    Returns  the IO mode of the IO line provided.

    Return type  IOMode

    Raises

    •  TimeoutException – if the response is not received before the read timeout expires.

    •  XBeeException – if the XBees serial port is closed.

    •  InvalidOperatingModeException – if the XBees operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

    •  ATCommandException – if the response is not as expected.

    •  OperationNotSupportedException – if the received data is not an IO mode.

get_io_sampling_rate()  
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

get_ip_addr()  
Returns the IP address of this IP device.

To refresh this value use the method IPDevice.read_device_info().

    Returns  The IP address of this IP device.

    Return type  ipaddress.IPv4Address

See also:

ipaddress.IPv4Address
get_network()  
Deprecated.  
This protocol does not support the network functionality.

get_next_frame_id()  
Returns the next frame ID of the XBee device.  

    Returns: The next frame ID of the XBee device.  
    Return type: Integer

get_node_id()  
Deprecated.  
Operation not supported in this protocol. This method will raise an AttributeError.

get_pan_id()  
Deprecated.  
Operation not supported in this protocol. This method will raise an AttributeError.

get_parameter(param, parameter_value=None)  
Override.  
See also:  

    AbstractXBeeDevice.get_parameter()

get_power_level()  
Deprecated.  
Operation not supported in this protocol. This method will raise an AttributeError.

get_protocol()  
Override.  
See also:  

    XBeeDevice.get_protocol()

get_pwm_duty_cycle(io_line)  
Returns the PWM duty cycle in % corresponding to the provided IO line.  

    Parameters: io_line (IOLine) – the IO line to get its PWM duty cycle.  
    Returns: the PWM duty cycle of the given IO line or None if the response is empty.  
    Return type: Integer  
    Raises:
        • TimeoutException – if the response is not received before the read timeout expires.  
        • XBeeException – if the XBee device’s serial port is closed.  
        • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.  
        • ATCommandException – if the response is not as expected.
• **ValueError** – if the passed IO_LINE has no PWM capability.

See also:

IOLine

**get_role()**

Gets the XBee role.

**Returns** the role of the XBee.

**Return type** `digi.xbee.models.protocol.Role`

See also:

`digi.xbee.models.protocol.Role`

**get_route_to_node** *(remote, timeout=10)*

Gets the route from this XBee to the given remote node.

**Parameters**

* remote (*RemoteXBeeDevice*) – The remote node.

* timeout (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.

**Returns**

Tuple containing route data *(None if the route was not read in the provided timeout):*

  - *source* (*RemoteXBeeDevice*): The source node of the route.
  - *destination* (*RemoteXBeeDevice*): The destination node of the route.

  * hops (*List*): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

**Return type** Tuple

**get_socket_info** *(socket_id)*

Returns the information of the socket with the given socket ID.

**Parameters** `socket_id` (*Integer*) – ID of the socket.

**Returns** The socket information, or None if the socket with that ID does not exist.

**Return type** `SocketInfo`

**Raises**

* `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

* `TimeoutException` – if the response is not received before the read timeout expires.

* `XBeeException` – if the XBee device’s serial port is closed.

See also:
SocketInfo

get_sockets_list()
Returns a list with the IDs of all active (open) sockets.

Returns list with the IDs of all active (open) sockets, or empty list if there is not any active socket.

Return type List

Raises
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns the serial port read timeout in seconds.

Return type Integer

get_xbee_device_callbacks()
Returns this XBee internal callbacks for process received packets.

This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

Returns PacketReceived

has_explicit_packets()
Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

Returns True if this XBee device’s queue has explicit packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()
Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

Returns True if this XBee device’s queue has packets, False otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()
is_apply_changes_enabled()  
Returns whether the apply_changes flag is enabled or not.  

Returns True if the apply_changes flag is enabled, False otherwise.  
Return type Boolean

is_connected()  
Returns whether the device is connected to the Internet or not.  

Returns True if the device is connected to the Internet, False otherwise.  
Return type Boolean  
Raises  
• TimeoutException – if there is a timeout getting the association indication status.  
• XBeeException – if there is any other XBee related exception.

is_open()  
Returns whether this XBee device is open or not.  

Returns Boolean. True if this XBee device is open, False otherwise.

is_remote()  
Override method.  
See also:  

AbstractXBeeDevice.is_remote()

log  
Returns the XBee device log.  

Returns the XBee device logger.  
Return type Logger

open(force_settings=False)  
Override.  

Raises  
• TimeoutException – If there is any problem with the communication.  
• InvalidOperatingModeException – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.  
• XBeeException – If the protocol is invalid or if the XBee device is already open.  
See also:  

XBeeDevice.open()

operating_mode  
Returns this XBee device’s operating mode.  

Returns OperatingMode. This XBee device’s operating mode.
reachable
Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

read_data (timeout=None, explicit=False)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

read_data_from (remote_xbee_device, timeout=None, explicit=False)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

read_device_info (init=True, fire_event=True)
Override.

See also:

XBeeDevice.read_device_info()

read_io_sample ()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type IOSample

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

See also:

IOSample

read_ip_data (timeout=3)
Reads new IP data received by this XBee device during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method IPDevice.add_ip_data_received_callback().

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method IPDevice.start_listening() for that purpose. When finished, you can use the method IPDevice.stop_listening() to stop listening for incoming IP data.
Parameters **timeout** *(Integer, optional)* – The time to wait for new IP data in seconds.

Returns **IP message**, **None** if this device did not receive new data.

Return type **IPMessage**

Raises **ValueError** – if **timeout** is less than 0.

**read_ip_data_from**(ip_addr, timeout=3)

Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method **IPDevice.add_ip_data_received_callback()**.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method **IPDevice.start_listening()** for that purpose. When finished, you can use the method **IPDevice.stop_listening()** to stop listening for incoming IP data.

Parameters

- **ip_addr** *(ipaddress.IPv4Address)* – The IP address to read data from.
- **timeout** *(Integer, optional)* – The time to wait for new IP data in seconds.

Returns **IP message**, **None** if this device did not receive new data from the provided IP address.

Return type **IPMessage**

Raises **ValueError** – if **timeout** is less than 0.

**reset**()

Override method.

See also:

AbstractXBeeDevice.reset()

**scan_counter**

Returns the scan counter for this node.

Returns **The scan counter for this node.**

Return type **Integer**

**sendBluetoothData**(data)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **data** *(Bytearray)* – Data to send.

Raises

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – if there is any problem sending the data.

See also:
send_data (remote_xbee_device, data, transmit_options=0)
  Deprecated.
  Operation not supported in this protocol. This method will raise an AttributeError.

send_data_async (remote_xbee_device, data, transmit_options=0)
  Deprecated.
  Operation not supported in this protocol. This method will raise an AttributeError.

send_data_broadcast (data, transmit_options=0)
  Deprecated.
  Operation not supported in this protocol. This method will raise an AttributeError.

send_ip_data_broadcast (dest_port, data)
  Sends the provided IP data to all clients.
  This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

  Parameters
  • dest_port (Integer) – The destination port of the transmission.
  • data (String or Bytearray) – The IP data to be sent.

  Raises
  • ValueError – if data is None.
  • ValueError – if dest_port is less than 0 or greater than 65535.
  • TimeoutException – if there is a timeout sending the data.
  • XBeeException – if there is any other XBee related exception.

send_micropython_data (data)
  Sends the given data to the MicroPython interface using a User Data Relay frame.

  Parameters data (Bytearray) – Data to send.

  Raises
  • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  • XBeeException – if there is any problem sending the data.

See also:

send_packet (packet, sync=False)
  Override method.

See also:
send_packet_sync_and_get_response \( packet\_to\_send, \text{timeout}=\text{None} \)

Override method.

See also:

AbstractXBeeDevice._send_packet_sync_and_get_response()
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the current protocol is not 802.15.4.

**set_api_output_mode** *(api_output_mode)*

Deprecated since version 1.3: Use **set_api_output_mode_value()**

Sets the API output mode of the XBee device.

**Parameters**

- **api_output_mode** *(APIOutputMode)* – the new API output mode of the XBee device.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.

- **OperationNotSupportedException** – if the current protocol is ZigBee

**See also:**

- **APIOutputMode**

**set_api_output_mode_value** *(api_output_mode)*

Sets the API output mode of the XBee.

**Parameters**

- **api_output_mode** *(Integer)* – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- **ATCommandException** – if the response is not as expected.

- **OperationNotSupportedException** – if it is not supported by the current protocol.

**See also:**

- `digi.xbee.models.mode.APIOutputModeBit`
**set_dest_address** (*addr*)

Deprecated.

Operation not supported in this protocol. Use `IPDevice.set_dest_ip_addr()` instead. This method will raise an `AttributeError`.

**set_dest_ip_addr** (*address*)

Sets the destination IP address.

**Parameters**

- **address** (*ipaddress.IPv4Address*) – Destination IP address.

**Raises**

- `ValueError` – if `address` is `None`.
- `TimeoutException` – if there is a timeout setting the destination IP address.
- `XBeeException` – if there is any other XBee related exception.

**See also:**

`ipaddress.IPv4Address`

**set_dio_change_detection** (*io_lines_set*)

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

**set_dio_value** (*io_line, io_value*)

Sets the digital value (high or low) to the provided IO line.

**Parameters**

- **io_line** (*IOLine*) – the digital IO line to sets its value.
- **io_value** (*IOValue*) – the IO value to set to the IO line.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**See also:**

`IOLine`
`IOValue`

**set_io_configuration** (*io_line, io_mode*)

Sets the configuration of the provided IO line.

**Parameters**

- **io_line** (*IOLine*) – the IO line to configure.
- **io_mode** (*IOMode*) – the IO mode to set to the IO line.
 Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

- `IOLine`
- `IOMode`

```python
set_io_sampling_rate(rate)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
set_node_id(node_id)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
set_pan_id(value)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
set_parameter(param, value)
```

Override.

See: `AbstractXBeeDevice.set_parameter()`

```python
set_power_level(power_level)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
set_pwm_duty_cycle(io_line, cycle)
```

Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

**Parameters**

- `io_line` (IOLine) – the IO Line to be assigned.
- `cycle` (Integer) – duty cycle in % to be assigned. Must be between 0 and 100.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
• **ValueError** – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

**See also:**

*IOLine*  
*IOMode.PWM*

### set_sync_ops_timeout(sync_ops_timeout)

Sets the serial port read timeout.

**Parameters**  
**sync_ops_timeout** (*Integer*) – the read timeout, expressed in seconds.

### start_listening(source_port)

Starts listening for incoming IP transmissions in the provided port.

**Parameters**  
**source_port** (*Integer*) – Port to listen for incoming transmissions.

**Raises**

• **ValueError** – if source_port is less than 0 or greater than 65535.

• **TimeoutException** – if there is a timeout setting the source port.

• **XBeeException** – if there is any other XBee related exception.

### stop_listening()

Stops listening for incoming IP transmissions.

**Raises**

• **TimeoutException** – if there is a timeout processing the operation.

• **XBeeException** – if there is any other XBee related exception.

### update_bluetooth_password(new_password)

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**  
**new_password** (*String*) – New Bluetooth password.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

### update_device_data_from(device)

Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**  
**device** (*AbstractXBeeDevice*) – the XBee device to get the data from.

**Returns**  
True if the device data has been updated, False otherwise.

**Return type**  
Boolean
update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)

Performs a firmware update operation of the device.

Parameters

- xml_firmware_file (String) – path of the XML file that describes the firmware to upload.
- xbee_firmware_file (String, optional) – location of the XBee binary firmware file.
- bootloader_firmware_file (String, optional) – location of the bootloader binary firmware file.
- timeout (Integer, optional) – the maximum time to wait for target read operations during the update process.
- progress_callback (Function, optional) – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- XBeeException – if the device is not open.
- InvalidOperatingModeException – if the device operating mode is invalid.
- OperationNotSupportedException – if the firmware update is not supported in the XBee device.
- FirmwareUpdateException – if there is any error performing the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method is_apply_configuration_changes_enabled() to get its status and enable_apply_configuration_changes() to enable/disable the option. If it is disabled, method apply_changes() can be used in order to manually apply the changes.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
class digi.xbee.devices.NBIoTDevice(port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None)

Bases: digi.xbee.devices.LPWANDevice

This class represents a local NB-IoT device.

Class constructor. Instantiates a new NBIoTDevice with the provided parameters.

Parameters

- **port** (Integer or String) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.
- **baud_rate** (Integer) – the serial port baud rate.
- (Integer, default (_sync_ops_timeout) – serial.EIGHTBITS): comm port bitsize.
- (Integer, default (serial.STOPBITS_ONE): comm port stop bits.
- (Character, default (parity) – serial.PARITY_NONE): comm port parity.
- (Integer, default (FlowControl.NONE): comm port flow control.
- (Integer, default (3): the read timeout (in seconds).
- **comm_iface** (XBeeCommunicationInterface) – the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor.

See also:

LPWANDevice

LPWANDevice.__init__()  

open (force_settings=False)

Override.

Raises

- **TimeoutException** – If there is any problem with the communication.
- **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – If the protocol is invalid or if the XBee device is already open.

See also:

XBeeDevice.open()  

get_protocol()

Override.
See also:

XBeeDevice.get_protocol()

**add_bluetooth_data_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.BluetoothDataReceived`.

Parameters **callback (Function)** – the callback. Receives one argument.
- The Bluetooth data as a Bytearray

**add_data_received_callback (callback)**
Deprecated.
Operation not supported in this protocol. This method will raise an `AttributeError`.

**add_expl_data_received_callback (callback)**
Deprecated.
Operation not supported in this protocol. This method will raise an `AttributeError`.

**add_io_sample_received_callback (callback)**
Deprecated.
Operation not supported in this protocol. This method will raise an `AttributeError`.

**add_ip_data_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.IPDataReceived`.

Parameters **callback (Function)** – the callback. Receives one argument.
- The data received as an `digi.xbee.models.message.IPMessage`

**add_micropython_data_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.MicroPythonDataReceived`.

Parameters **callback (Function)** – the callback. Receives one argument.
- The MicroPython data as a Bytearray

**add_modem_status_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.ModemStatusReceived`.

Parameters **callback (Function)** – the callback. Receives one argument.
- The modem status as a `digi.xbee.models.status.ModemStatus`

**add_packet_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.PacketReceived`.

Parameters **callback (Function)** – the callback. Receives two arguments.
- The received packet as a `digi.xbee.packets.base.XBeeAPIPacket`

**add_route_received_callback (callback)**
Adds a callback for the event `RouteReceived`. This works for Zigbee and Digimesh devices.

Parameters **callback (Function)** – the callback. Receives three arguments.
- source (`XBeeDevice`): The source node.
- destination (`RemoteXBeeDevice`): The destination node.
• **hops (List):** List of intermediate hops from closest to source to closest to destination ([RemoteXBeeDevice](#)).

See also:

`XBeeDevice.del_route_received_callback()`

**add_sms_callback (callback)**
Depreciated.
Operation not supported in this protocol. This method will raise an `AttributeError`.

**add_socket_data_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.SocketDataReceived`.

**Parameters callback (Function)** – the callback. Receives two arguments.

• The socket ID as an Integer.

• The data received as Bytearray

**add_socket_data_received_from_callback (callback)**
Adds a callback for the event `digi.xbee.reader.SocketDataReceivedFrom`.

**Parameters callback (Function)** – the callback. Receives three arguments.

• The socket ID as an Integer.

• A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.

• The data received as Bytearray

**add_socket_state_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.SocketStateReceived`.

**Parameters callback (Function)** – the callback. Receives two arguments.

• The socket ID as an Integer.

• The state received as a `SocketState`

**add_user_data_relay_received_callback (callback)**
Adds a callback for the event `digi.xbee.reader.RelayDataReceived`.

**Parameters callback (Function)** – the callback. Receives one argument.

• The relay data as a `digi.xbee.models.message.UserDataRelayMessage`

**apply_changes ()**
Applies changes via AC command.

**Raises**

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.
apply_profile(profile_path, progress_callback=None)
Applies the given XBee profile to the XBee device.

Parameters

- **profile_path** *(String)* – path of the XBee profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **UpdateProfileException** – if there is any error applying the XBee profile.
- **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

close()
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

comm_iface
Returns the hardware interface associated to the XBee device.

Returns the hardware interface associated to the XBee device.

Return type **XBeeCommunicationInterface**

See also:

XBeeSerialPort
classmethod create_xbee_device(comm_port_data)
Creates and returns an XBeeDevice from data of the port to which is connected.

Parameters

- **comm_port_data** *(Dictionary)* – dictionary with all comm port data needed.

  - **dictionary keys are** *(The)* –
    - “baudRate” → Baud rate.
    - “port” → Port number.
    - “bitSize” → Bit size.
    - “stopBits” → Stop bits.
    - “parity” → Parity.
    - “flowControl” → Flow control.
    - “timeout” for → Timeout for synchronous operations (in seconds).

Returns the XBee device created.
Return type `XBeeDevice`

Raises `SerialException` – if the port you want to open does not exist or is already opened.

See also:

`XBeeDevice`

del_bluetooth_data_received_callback `(callback)`
Deletes a callback for the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

Parameters `callback` (`Function`) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

del_data_received_callback `(callback)`
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

del_expl_data_received_callback `(callback)`
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

del_io_sample_received_callback `(callback)`
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

del_ip_data_received_callback `(callback)`
Deletes a callback for the callback list of `digi.xbee.reader.IPDataReceived` event.

Parameters `callback` (`Function`) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.IPDataReceived` event.

del_micropython_data_received_callback `(callback)`
Deletes a callback for the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

Parameters `callback` (`Function`) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

del_modem_status_received_callback `(callback)`
Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

Parameters `callback` (`Function`) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

del_packet_received_callback `(callback)`
Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

Parameters `callback` (`Function`) – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.PacketReceived` event.
del_route_received_callback (callback)
Deletes a callback for the callback list of RouteReceived event.

Parameters callback (Function) – the callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_sms_callback (callback)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

del_socket_data_received_callback (callback)
Deletes a callback for the callback list of digi.xbee.reader.SocketDataReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of digi.xbee.reader. SocketDataReceived event.

del_socket_data_received_from_callback (callback)
Deletes a callback for the callback list of digi.xbee.reader.SocketDataReceivedFrom event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of digi.xbee.reader. SocketDataReceivedFrom event.

del_socket_state_received_callback (callback)
Deletes a callback for the callback list of digi.xbee.reader.SocketStateReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of digi.xbee.reader. SocketStateReceived event.

del_user_data_relay_received_callback (callback)
Deletes a callback for the callback list of digi.xbee.reader.RelayDataReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of digi.xbee.reader. RelayDataReceived event.

disable_bluetooth()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

enable_apply_changes (value)
Sets the apply_changes flag.
Parameters value (Boolean) – True to enable the apply changes flag, False to disable it.

enable_bluetooth ()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the AbstractXBeeDevice.update_bluetooth_password() method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

 Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

execute_command (parameter)
Executes the provided command.

Parameters parameter (String) – The name of the AT command to be executed.

 Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

flush_queues ()
Flushes the packets queue.

get_16bit_addr ()
Deprecated.
This protocol does not have an associated 16-bit address.

get_64bit_addr ()
Deprecated.
Cellular protocol does not have an associated 64-bit address.

get_adc_value (io_line)
Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.set_io_configuration() and IOMode.ADC.

Parameters io_line (IOLine) – the IO line to get its ADC value.

Returns the analog value corresponding to the provided IO line.

Return type Integer

 Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
XBee Python Library Documentation, Release 1.3.0

- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

`IOLine`

### get_api_output_mode()

Deprecated since version 1.3: Use `get_api_output_mode_value()`

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

**Returns** the API output mode of the XBee device.

**Return type** `APIOutputMode`

**Raises**
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

`APIOutputMode`

### get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

**Returns** the parameter value.

**Return type** `Bytearray`

**Raises**
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBees device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:
get_bluetooth_mac_addr()  
Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

Returns  The Bluetooth MAC address.
Return type  String
Raisers

•  

•  

•  

get_cellular_ai_status()  
Returns the current association status of this Cellular device.

It indicates occurrences of errors during the modem initialization and connection.

Returns  The association indication status of the Cellular device.
Return type  CellularAssociationIndicationStatus
Raisers

•  

•  

get_current_frame_id()  
Returns the last used frame ID.

Returns  the last used frame ID.
Return type  Integer

get_dest_address()  
Deprecated.

Operation not supported in this protocol. Use IDevice.get_dest_ip_addr() instead. This method will raise an AttributeError.

get_dest_ip_addr()  
Returns the destination IP address.

Returns  The configured destination IP address.
Return type  ipaddress.IPv4Address
Raisers

•  

•  

See also:

ipaddress.IPv4Address
get_dio_value(io_line)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use AbstractXBeeDevice.set_io_configuration().

Parameters io_line (IOLine) – the DIO line to get its digital value.

Returns current value of the provided IO line.

Return type IOValue

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the response does not contain the value for the given IO line.

See also:

IOLine
IOValue

get_firmware_version()
Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type HardwareVersion

See also:

HardwareVersion

get_imei_addr()
Returns the IMEI address of this Cellular device.

To refresh this value use the method CellularDevice.read_device_info().

Returns The IMEI address of this Cellular device.

Return type XBeeIMEIAddress

get_io_configuration(io_line)
Returns the configuration of the provided IO line.
**Parameters**  
*io_line* (*IOLine*) – the io line to configure.

**Returns**  
the IO mode of the IO line provided.

**Return type**  
*IOMode*

**Raises**

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.
- *OperationNotSupportedException* – if the received data is not an IO mode.

**get_io_sampling_rate()**

Deprecated.

Operation not supported in this protocol. This method will raise an *AttributeError*.

**get_ip_addr()**

Returns the IP address of this IP device.

To refresh this value use the method `IPDevice.read_device_info()`.

**Returns**  
The IP address of this IP device.

**Return type**  
*ipaddress.IPv4Address*

See also:

*ipaddress.IPv4Address*

**get_network()**

Deprecated.

This protocol does not support the network functionality.

**get_next_frame_id()**

Returns the next frame ID of the XBee device.

**Returns**  
The next frame ID of the XBee device.

**Return type**  
*Integer*

**get_node_id()**

Deprecated.

Operation not supported in this protocol. This method will raise an *AttributeError*.

**get_pan_id()**

Deprecated.

Operation not supported in this protocol. This method will raise an *AttributeError*.

**get_parameter** (*param, parameter_value=None*)

Override.

See also:
AbstractXBeeDevice.get_parameter()

get_power_level()

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

get_pwm_duty_cycle(io_line)

Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters io_line (IOLine) – the IO line to get its PWM duty cycle.

Returns the PWM duty cycle of the given IO line or None if the response is empty.

Return type Integer

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- ValueError – if the passed IO_LINE has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type digi.xbee.models.protocol.Role

See also:

digi.xbee.models.protocol.Role

get_route_to_node(remote, timeout=10)

Gets the route from this XBee to the given remote node.

Parameters

- remote (RemoteXBeeDevice) – The remote node.
- timeout (Float, optional, default=10) – Maximum number of seconds to wait for the route.

Returns

Tuple containing route data (None if the route was not read in the provided timeout):
- source (RemoteXBeeDevice): The source node of the route.
- destination (RemoteXBeeDevice): The destination node
of the route.

- hops (List): List of intermediate nodes (RemoteXBeeDevice) ordered from closest to source to closest to destination node (source and destination not included).

Return type: Tuple

get_socket_info(socket_id)
Returns the information of the socket with the given socket ID.

Parameters:
socket_id (Integer) – ID of the socket.

Returns: The socket information, or None if the socket with that ID does not exist.

Return type: SocketInfo

Raises:
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.

See also:

SocketInfo

get_sockets_list()
Returns a list with the IDs of all active (open) sockets.

Returns: list with the IDs of all active (open) sockets, or empty list if there is not any active socket.

Return type: List

Raises:
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns: the serial port read timeout in seconds.

Return type: Integer

get_xbee_device_callbacks()
Returns this XBee internal callbacks for process received packets.

This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.

Returns: PacketReceived
has_explicit_packets()

Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.

- **Returns**: True if this XBee device’s queue has explicit packets, False otherwise.
- **Return type**: Boolean

See also:

`XBeeDevice.has_packets()`

has_packets()

Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.

- **Returns**: True if this XBee device’s queue has packets, False otherwise.
- **Return type**: Boolean

See also:

`XBeeDevice.has_explicit_packets()`

is_apply_changes_enabled()

Returns whether the apply_changes flag is enabled or not.

- **Returns**: True if the apply_changes flag is enabled, False otherwise.
- **Return type**: Boolean

is_connected()

Returns whether the device is connected to the Internet or not.

- **Returns**: True if the device is connected to the Internet, False otherwise.
- **Return type**: Boolean

Raises

- **TimeoutException** – if there is a timeout getting the association indication status.
- **XBeeException** – if there is any other XBee related exception.

is_open()

Returns whether this XBee device is open or not.

- **Returns**: Boolean. True if this XBee device is open, False otherwise.

is_remote()

Override method.

See also:

`AbstractXBeeDevice.is_remote()`

log

Returns the XBee device log.
Returns the XBee device logger.

Return type Logger

**operating_mode**

Returns this XBee device’s operating mode.

Returns OperatingMode. This XBee device’s operating mode.

**reachable**

Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

**read_data** *(timeout=None, explicit=False)*

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

**read_data_from** *(remote_xbee_device, timeout=None, explicit=False)*

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

**read_device_info** *(init=True, fire_event=True)*

Override.

See also:

XBeeDevice.read_device_info()

**read_io_sample**()

Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type IOSample

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

IOSample

**read_ip_data** *(timeout=3)*

Reads new IP data received by this XBee device during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.
For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

### Parameters

- **timeout** `(Integer, optional)` – The time to wait for new IP data in seconds.

### Returns

IP message, `None` if this device did not receive new data.

### Return type

`IPMessage`

### Raises

`ValueError` – if `timeout` is less than 0.

### read_ip_data_from(ip_addr, timeout=3)

Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

### Parameters

- **ip_addr** `(ipaddress.IPv4Address)` – The IP address to read data from.
- **timeout** `(Integer, optional)` – The time to wait for new IP data in seconds.

### Returns

IP message, `None` if this device did not receive new data from the provided IP address.

### Return type

`IPMessage`

### Raises

`ValueError` – if `timeout` is less than 0.

### reset()

Override method.

See also:

AbstractXBeeDevice.reset()

### scan_counter

Returns the scan counter for this node.

### Returns

The scan counter for this node.

### Return type

`Integer`

### send_bluetooth_data(data)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

### Parameters

- **data** `(Bytearray)` – Data to send.

### Raises
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – if there is any problem sending the data.

See also:

```
XBeeDevice.send_micropython_data()
XBeeDevice.send_user_data_relay()
```

```python
send_data (remote_xbee_device, data, transmit_options=0)
```
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
send_data_async (remote_xbee_device, data, transmit_options=0)
```
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
send_data_broadcast (data, transmit_options=0)
```
Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
send_ip_data (ip_addr, dest_port, protocol, data, close_socket=False)
```
Sends the provided IP data to the given IP address and port using the specified IP protocol.

This method blocks till a success or error response arrives or the configured receive timeout expires.

**Parameters**

- **ip_addr** (ipaddress.IPv4Address) – The IP address to send IP data to.
- **dest_port** (Integer) – The destination port of the transmission.
- **protocol** (IPProtocol) – The IP protocol used for the transmission.
- **data** (String or bytearray) – The IP data to be sent.
- **close_socket** (Boolean, optional) – Must be `False`.

**Raises** `ValueError` – if `protocol` is not UDP.

```python
send_ip_data_async (ip_addr, dest_port, protocol, data, close_socket=False)
```
Sends the provided IP data to the given IP address and port asynchronously using the specified IP protocol.

Asynchronous transmissions do not wait for answer from the remote device or for transmit status packet.

**Parameters**

- **ip_addr** (ipaddress.IPv4Address) – The IP address to send IP data to.
- **dest_port** (Integer) – The destination port of the transmission.
- **protocol** (IPProtocol) – The IP protocol used for the transmission.
- **data** (String or bytearray) – The IP data to be sent.
- **close_socket** (Boolean, optional) – Must be `False`.

**Raises** `ValueError` – if `protocol` is not UDP.
send_ip_data_broadcast \((dest\_port, data)\)
Sends the provided IP data to all clients.
This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

Parameters

- **dest_port** \((\text{Integer})\) – The destination port of the transmission.
- **data** \((\text{String or Bytearray})\) – The IP data to be sent.

Raises

- **ValueError** – if data is None.
- **ValueError** – if dest_port is less than 0 or greater than 65535.
- **TimeoutException** – if there is a timeout sending the data.
- **XBeeException** – if there is any other XBee related exception.

send_micropython_data \((data)\)
Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters **data** \((\text{Bytearray})\) – Data to send.

Raises

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **XBeeException** – if there is any problem sending the data.

See also:

- `XBeeDevice.send_bluetooth_data()`  
- `XBeeDevice.send_user_data_relay()`

send_packet \((packet, sync=False)\)
Override method.

See also:

- `AbstractXBeeDevice._send_packet()`

send_packet_sync_and_get_response \((packet\_to\_send, timeout=None)\)
Override method.

See also:

- `AbstractXBeeDevice._send_packet_sync_and_get_response()`

send_sms \((phone\_number, data)\)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.
send_sms_async(phone_number, data)
Depreciated.
Operation not supported in this protocol. This method will raise an AttributeError.

send_user_data_relay(local_interface, data)
Sends the given data to the given XBee local interface.

Parameters
- local_interface (XBeeLocalInterface) – Destination XBee local interface.
- data (Bytearray) – Data to send.

Raises
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ValueError – if local_interface is None.
- XBeeException – if there is any problem sending the User Data Relay.

See also:
XBeeLocalInterface

serial_port
Returns the serial port associated to the XBee device, if any.

Returns
the serial port associated to the XBee device. Returns None if the local XBee does not use serial communication.

Return type XBeeSerialPort

See also:
XBeeSerialPort

set_16bit_addr(value)
Sets the 16-bit address of the XBee device.

Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the current protocol is not 802.15.4.
set_api_output_mode (api_output_mode)
Depreciated since version 1.3: Use set_api_output_mode_value()

Sets the API output mode of the XBee device.

Parameters api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the current protocol is ZigBee

See also:

APIOutputMode

set_api_output_mode_value (api_output_mode)

Sets the API output mode of the XBee.

Parameters api_output_mode (Integer) – new API output mode options. Calculate this value using the method digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value() with a set of digi.xbee.models.mode.APIOutputModeBit.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

set_dest_address (addr)

Deprecated.

Operation not supported in this protocol. Use IDevice.set_dest_ip_addr() instead. This method will raise an AttributeError.

set_dest_ip_addr (address)

Sets the destination IP address.
Parameters **address** (ipaddress.IPv4Address) – Destination IP address.

Raises

- **ValueError** – if address is None.
- **TimeoutException** – if there is a timeout setting the destination IP address.
- **XBeeException** – if there is any other XBee related exception.

See also:

ipaddress.IPv4Address

**set_dio_change_detection** (*io_lines_set*)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

**set_dio_value** (*io_line, io_value*)

Sets the digital value (high or low) to the provided IO line.

Parameters

- **io_line** (*IOLine*) – the digital IO line to sets its value.
- **io_value** (*IOValue*) – the IO value to set to the IO line.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

*IOLine*  
*IOValue*

**set_io_configuration** (*io_line, io_mode*)

Sets the configuration of the provided IO line.

Parameters

- **io_line** (*IOLine*) – the IO line to configure.
- **io_mode** (*IOMode*) – the IO mode to set to the IO line.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
set_io_sampling_rate(rate)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

set_node_id(node_id)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

set_pan_id(value)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

set_parameter(param, value)

Override.

See: AbstractXBeeDevice.set_parameter()

set_power_level(power_level)

Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

set_pwm_duty_cycle(io_line, cycle)

Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

- io_line(IOLine) – the IO Line to be assigned.
- cycle(Integer) – duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- ValueError – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

IOLine
IOMode.PWM
set_sync_ops_timeout(sync_ops_timeout)
Sets the serial port read timeout.

Parameters
 sync_ops_timeout (Integer) – the read timeout, expressed in seconds.

start_listening(source_port)
Starts listening for incoming IP transmissions in the provided port.

Parameters
 source_port (Integer) – Port to listen for incoming transmissions.

Raises
 • ValueError – if source_port is less than 0 or greater than 65535.
 • TimeoutException – if there is a timeout setting the source port.
 • XBeeException – if there is any other XBee related exception.

stop_listening()
Stops listening for incoming IP transmissions.

Raises
 • TimeoutException – if there is a timeout processing the operation.
 • XBeeException – if there is any other XBee related exception.

update_bluetooth_password(new_password)
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

Parameters
 new_password (String) – New Bluetooth password.

Raises
 • TimeoutException – if the response is not received before the read timeout expires.
 • XBeeException – if the XBee device’s serial port is closed.
 • InvalidOperatingModeException – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

update_device_data_from(device)
Updates the current device reference with the data provided for the given device.

This is only for internal use.

Parameters
 device (AbstractXBeeDevice) – the XBee device to get the data from.

Returns True if the device data has been updated, False otherwise.

Return type Boolean

update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)
Performs a firmware update operation of the device.

Parameters
 • xml_firmware_file (String) – path of the XML file that describes the firmware to upload.
 • xbee_firmware_file (String, optional) – location of the XBee binary firmware file.
 • bootloader_firmware_file (String, optional) – location of the bootloader binary firmware file.
• **timeout** (*Integer, optional*) – the maximum time to wait for target read operations during the update process.

• **progress_callback** (*Function, optional*) – function to execute to receive progress information. Receives two arguments:
  
  – The current update task as a String
  – The current update task percentage as an Integer

**Raises**

• **XBeeException** – if the device is not open.

• **InvalidOperatingModeException** – if the device operating mode is invalid.

• **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.

• **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes()**

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method. If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method `is_apply_configuration_changes_enabled()` to get its status and `enable_apply_configuration_changes()` to enable/disable the option. If it is disabled, method `apply_changes()` can be used in order to manually apply the changes.

**Raises**

• **FormatException** – if the response is not as expected.

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

**class** **digi.xbee.devices.WiFiDevice**

```python
(port=port, data_bits=data_bits, stop_bits=stop_bits, parity=parity, flow_control=flow_control,
_baud_rate=baud_rate, _sync_ops_timeout=_sync_ops_timeout, comm_iface=comm_iface)
```

**Bases:** **digi.xbee.devices.IPDevice**

This class represents a local Wi-Fi XBee device.

Class constructor. Instantiates a new WiFiDevice with the provided parameters.

**Parameters**

• **port** (*Integer or String*) – serial port identifier. Integer: number of XBee device, numbering starts at zero. Device name: depending on operating system. e.g. ‘/dev/ttyUSB0’ on ‘GNU/Linux’ or ‘COM3’ on Windows.
• **baud_rate** (*Integer*) – the serial port baud rate.

• *(Integer, default \(_\text{sync}_{-}\text{ops}_{-}\text{timeout}\) – \(\text{serial.\text{EIGHTBITS}}\))*: comm port bitsize.

• *(Integer, default \(\text{serial.\text{STOPBITS\_ONE}}\))*: comm port stop bits.

• *(Character, default \(\text{serial.\text{PARITY\_NONE}}\))*: comm port parity.

• *(Integer, default \(\text{FlowControl.\text{NONE}}\))*: comm port flow control.

• *(Integer, default \(3\))*: the read timeout (in seconds).

• **comm_iface** (*XBeeCommunicationInterface*) – the communication interface.

:raises All exceptions raised by XBeeDevice.__init__() constructor.

See also:

```python
IPDevice
v.__init__()
```

**open** *(force_settings=False)*

Override.

Raises

• **TimeoutException** – If there is any problem with the communication.

• **InvalidOperatingModeException** – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **XBeeException** – If the protocol is invalid or if the XBee device is already open.

See also:

```python
XBeeDevice.open()
```

**get_protocol**

Override.

See also:

```python
XBeeDevice.get_protocol()
```

**get_wifi_ai_status**

Returns the current association status of the device.

Returns the current association status of the device.

Return type **WiFiAssociationIndicationStatus**

Raises

• **TimeoutException** – if there is a timeout getting the association indication status.

• **XBeeException** – if there is any other XBee related exception.
See also:

*WiFiAssociationIndicationStatus*

**`get_access_point(ssid)`**

Finds and returns the access point that matches the supplied SSID.

**Parameters** `ssid(String)` – the SSID of the access point to get.

**Returns**

the discovered access point with the provided SSID, or `None` if the timeout expires and the access point was not found.

**Return type** `AccessPoint`

**Raises**

- `TimeoutException` – if there is a timeout getting the access point.
- `XBeeException` – if there is an error sending the discovery command.

See also:

*AccessPoint*

**`scan_access_points()`**

Performs a scan to search for access points in the vicinity.

This method blocks until all the access points are discovered or the configured access point timeout expires.

The access point timeout is configured using the `WiFiDevice.set_access_point_timeout()` method and can be consulted with `WiFiDevice.get_access_point_timeout()` method.

**Returns** the list of `AccessPoint` objects discovered.

**Return type** `List`

**Raises**

- `TimeoutException` – if there is a timeout scanning the access points.
- `XBeeException` – if there is any other XBee related exception.

See also:

*AccessPoint*

**`connect_by_ap(access_point, password=None)`**

Connects to the provided access point.

This method blocks until the connection with the access point is established or the configured access point timeout expires.

The access point timeout is configured using the `WiFiDevice.set_access_point_timeout()` method and can be consulted with `WiFiDevice.get_access_point_timeout()` method.
Once the module is connected to the access point, you can issue the `WiFiDevice.write_changes()` method to save the connection settings. This way the module will try to connect to the access point every time it is powered on.

**Parameters**

- **access_point** (*AccessPoint*) – The access point to connect to.
- **password** (*String, optional*) – The password for the access point, `None` if it does not have any encryption enabled. Optional.

**Returns**  
`True` if the module connected to the access point successfully, `False` otherwise.

**Return type**  
`Boolean`

**Raises**

- `ValueError` – if `access_point` is `None`.
- `TimeoutException` – if there is a timeout sending the connect commands.
- `XBeeException` – if there is any other XBee related exception.

**See also:**

- `WiFiDevice.connect_by_ssid()`  
- `WiFiDevice.disconnect()`  
- `WiFiDevice.get_access_point()`  
- `WiFiDevice.get_access_point_timeout()`  
- `WiFiDevice.scan_access_points()`  
- `WiFiDevice.set_access_point_timeout()`

```python
connect_by_ssid(ssid, password=None)
```

Connects to the access point with provided SSID.

This method blocks until the connection with the access point is established or the configured access point timeout expires.

The access point timeout is configured using the `WiFiDevice.set_access_point_timeout()` method and can be consulted with `WiFiDevice.get_access_point_timeout()` method.

Once the module is connected to the access point, you can issue the `WiFiDevice.write_changes()` method to save the connection settings. This way the module will try to connect to the access point every time it is powered on.

**Parameters**

- **ssid** (*String*) – the SSID of the access point to connect to.
- **password** (*String, optional*) – The password for the access point, `None` if it does not have any encryption enabled. Optional.

**Returns**  
`True` if the module connected to the access point successfully, `False` otherwise.

**Return type**  
`Boolean`

**Raises**

- `ValueError` – if `ssid` is `None`.
- `TimeoutException` – if there is a timeout sending the connect commands.
• **XBeeException** – if the access point with the provided SSID cannot be found.
• **XBeeException** – if there is any other XBee related exception.

See also:

```python
WiFiDevice.connect_by_ap()
WiFiDevice.disconnect()
WiFiDevice.get_access_point()
WiFiDevice.get_access_point_timeout()
WiFiDevice.scan_access_points()
WiFiDevice.set_access_point_timeout()
```

### disconnect()

Disconnects from the access point that the device is connected to.

This method blocks until the device disconnects totally from the access point or the configured access point timeout expires.

The access point timeout is configured using the `WiFiDevice.set_access_point_timeout()` method and can be consulted with `WiFiDevice.get_access_point_timeout()` method.

- **Returns** `True` if the module disconnected from the access point successfully, `False` otherwise.
- **Return type** Boolean
- **Raises**
  - **TimeoutException** – if there is a timeout sending the disconnect command.
  - **XBeeException** – if there is any other XBee related exception.

See also:

```python
WiFiDevice.connect_by_ap()
WiFiDevice.connect_by_ssid()
WiFiDevice.get_access_point_timeout()
WiFiDevice.set_access_point_timeout()
```

### is_connected()

Returns whether the device is connected to an access point or not.

- **Returns** `True` if the device is connected to an access point, `False` otherwise.
- **Return type** Boolean
- **Raises** **TimeoutException** – if there is a timeout getting the association indication status.

See also:

```python
WiFiDevice.get_wifi_ai_status()
WiFiAssociationIndicationStatus
```
get_access_point_timeout()
Returns the configured access point timeout for connecting, disconnecting and scanning access points.

Returns the current access point timeout in milliseconds.
Return type Integer
See also:
WiFiDevice.set_access_point_timeout()

set_access_point_timeout(ap_timeout)
Configures the access point timeout in milliseconds for connecting, disconnecting and scanning access points.

Parameters ap_timeout (Integer) – the new access point timeout in milliseconds.
Raises ValueError – if ap_timeout is less than 0.
See also:
WiFiDevice.get_access_point_timeout()

get_ip_addressing_mode()
Returns the IP addressing mode of the device.

Returns the IP addressing mode.
Return type IPAddressingMode
Raises TimeoutException – if there is a timeout reading the IP addressing mode.
See also:
WiFiDevice.set_ip_addressing_mode()
IPAddressingMode

set_ip_addressing_mode(mode)
Sets the IP addressing mode of the device.

Parameters mode (IPAddressingMode) – the new IP addressing mode to set.
Raises TimeoutException – if there is a timeout setting the IP addressing mode.
See also:
WiFiDevice.get_ip_addressing_mode()
IPAddressingMode

set_ip_address(ip_address)
Sets the IP address of the module.
This method can only be called if the module is configured in IPAddressingMode.STATIC mode. Otherwise an XBeeException will be thrown.

**Parameters** `ip_address` (ipaddress.IPv4Address) – the new IP address to set.

**Raises** `TimeoutException` – if there is a timeout setting the IP address.

See also:

```python
WiFiDevice.get_mask_address()
ipaddress.IPv4Address
```

**get_mask_address()**

Returns the subnet mask IP address.

**Returns** the subnet mask IP address.

**Return type** `ipaddress.IPv4Address`

**Raises** `TimeoutException` – if there is a timeout reading the subnet mask address.

See also:

```python
WiFiDevice.set_mask_address()
```

```python
ipaddress.IPv4Address
```

**set_mask_address(mask_address)**

Sets the subnet mask IP address.

This method can only be called if the module is configured in IPAddressingMode.STATIC mode. Otherwise an XBeeException will be thrown.

**Parameters** `mask_address` (ipaddress.IPv4Address) – the new subnet mask address to set.

**Raises** `TimeoutException` – if there is a timeout setting the subnet mask address.

See also:

```python
WiFiDevice.get_mask_address()
```

```python
ipaddress.IPv4Address
```

**get_gateway_address()**

Returns the IP address of the gateway.

**Returns** the IP address of the gateway.

**Return type** `ipaddress.IPv4Address`

**Raises** `TimeoutException` – if there is a timeout reading the gateway address.

See also:

```python
WiFiDevice.set_dns_address()
```
`set_gateway_address(gateway_address)`
Sets the IP address of the gateway.
This method can only be called if the module is configured in `IPAddressingMode.STATIC` mode. Otherwise an `XBeeException` will be thrown.

**Parameters**
`gateway_address` (ipaddress.IPv4Address) – the new gateway address to set.

**Raises** `TimeoutException` – if there is a timeout setting the gateway address.

See also:

```
WiFiDevice.get_gateway_address()
```

`get_dns_address()`
Returns the IP address of Domain Name Server (DNS).

**Returns**
the DNS address configured.

**Return type**
ipaddress.IPv4Address

**Raises** `TimeoutException` – if there is a timeout reading the DNS address.

See also:

```
WiFiDevice.set_dns_address()
```

`set_dns_address(dns_address)`
Sets the IP address of Domain Name Server (DNS).

**Parameters**
`dns_address` (ipaddress.IPv4Address) – the new DNS address to set.

**Raises** `TimeoutException` – if there is a timeout setting the DNS address.

See also:

```
WiFiDevice.get_dns_address()
```

`add_bluetooth_data_received_callback(callback)`
Adds a callback for the event `digi.xbee.reader.BluetoothDataReceived`.

**Parameters**
`callback` (Function) – the callback. Receives one argument.

- The Bluetooth data as a Bytearray
add_data_received_callback (callback)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

add_expl_data_received_callback (callback)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

add_io_sample_received_callback (callback)
Adds a callback for the event digi.xbee.reader.IOSampleReceived.

Parameters callback (Function) – the callback. Receives three arguments.
- The received IO sample as an digi.xbee.io.IOSample
- The remote XBee device who has sent the packet as a RemoteXBeeDevice
- The time in which the packet was received as an Integer

add_ip_data_received_callback (callback)
Adds a callback for the event digi.xbee.reader.IPDataReceived.

Parameters callback (Function) – the callback. Receives one argument.
- The data received as an digi.xbee.models.message.IPMessage

add_micropython_data_received_callback (callback)
Adds a callback for the event digi.xbee.reader.MicroPythonDataReceived.

Parameters callback (Function) – the callback. Receives one argument.
- The MicroPython data as a Bytearray

add_modem_status_received_callback (callback)
Adds a callback for the event digi.xbee.reader.ModemStatusReceived.

Parameters callback (Function) – the callback. Receives one argument.
- The modem status as a digi.xbee.models.status.ModemStatus

add_packet_received_callback (callback)
Adds a callback for the event digi.xbee.reader.PacketReceived.

Parameters callback (Function) – the callback. Receives two arguments.
- The received packet as a digi.xbee.packets.base.XBeeAPIPacket

add_route_received_callback (callback)
Adds a callback for the event RouteReceived. This works for Zigbee and Digimesh devices.

Parameters callback (Function) – the callback. Receives three arguments.
- source (XBeeDevice): The source node.
- destination (RemoteXBeeDevice): The destination node.
- hops (List): List of intermediate hops from closest to source to closest to destination (RemoteXBeeDevice).

See also:

XBeedevicendelete_route_received_callback()
add_socket_data_received_callback(callback)
Adds a callback for the event `digio.xbee.reader.SocketDataReceived`.

Parameters **callback** *(Function)* – the callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray

add_socket_data_received_from_callback(callback)
Adds a callback for the event `digio.xbee.reader.SocketDataReceivedFrom`.

Parameters **callback** *(Function)* – the callback. Receives three arguments.

- The socket ID as an Integer.
- A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
- The data received as Bytearray

add_socket_state_received_callback(callback)
Adds a callback for the event `digio.xbee.reader.SocketStateReceived`.

Parameters **callback** *(Function)* – the callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a `SocketState`

add_user_data_relay_received_callback(callback)
Adds a callback for the event `digio.xbee.reader.RelayDataReceived`.

Parameters **callback** *(Function)* – the callback. Receives one argument.

- The relay data as a `digio.xbee.models.message.UserDataRelayMessage`

apply_changes()
Applies changes via AC command.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

apply_profile(profile_path, progress_callback=None)
Applies the given XBee profile to the XBee device.

Parameters

- **profile_path** *(String)* – path of the XBee profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises
• **XBeeException** – if the device is not open.
• **InvalidOperatingModeException** – if the device operating mode is invalid.
• **UpdateProfileException** – if there is any error applying the XBee profile.
• **OperationNotSupportedException** – if XBee profiles are not supported in the XBee device.

**close()**
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

**comm_iface**
Returns the hardware interface associated to the XBee device.

Returns the hardware interface associated to the XBee device.

Return type **XBeeCommunicationInterface**

See also:

**XBeeSerialPort**

**classmethod create_xbee_device**(comm_port_data)
Creates and returns an **XBeeDevice** from data of the port to which is connected.

Parameters

• **comm_port_data** *(Dictionary)* – dictionary with all comm port data needed.

• **dictionary keys are (The)** –

  “baudRate” -> Baud rate.
  "port" -> Port number.
  "bitSize" -> Bit size.
  "stopBits" -> Stop bits.
  "parity" -> Parity.
  "flowControl" -> Flow control.
  "timeout" for -> Timeout for synchronous operations (in seconds).

Returns the XBee device created.

Return type **XBeeDevice**

Raises **SerialException** – if the port you want to open does not exist or is already opened.

See also:

**XBeeDevice**

**del_bluetooth_data_received_callback**(callback)
Deletes a callback for the callback list of **digi.xbee.reader.BluetoothDataReceived** event.

Parameters **callback** *(Function)* – the callback to delete.
Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.BluetoothDataReceived` event.

```python
def del_data_received_callback(callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
def del_expl_data_received_callback(callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
def del_io_sample_received_callback(callback)
```

Deletes a callback for the callback list of `digi.xbee.reader.IOSampleReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.IOSampleReceived` event.

```python
def del_ip_data_received_callback(callback)
```

Deletes a callback for the callback list of `digi.xbee.reader.IPDataReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.IPDataReceived` event.

```python
def del_micropython_data_received_callback(callback)
```

Deletes a callback for the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.MicroPythonDataReceived` event.

```python
def del_modem_status_received_callback(callback)
```

Deletes a callback for the callback list of `digi.xbee.reader.ModemStatusReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.ModemStatusReceived` event.

```python
def del_packet_received_callback(callback)
```

Deletes a callback for the callback list of `digi.xbee.reader.PacketReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

Raises `ValueError` – if `callback` is not in the callback list of `digi.xbee.reader.PacketReceived` event.

```python
def del_route_received_callback(callback)
```

Deletes a callback for the callback list of `RouteReceived` event.

Parameters `callback` *(Function)* – the callback to delete.

See also:

```python
XBeeDevice.add_route_received_callback()
```
**del_socket_data_received_callback** *(callback)*
 Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceived` event.

**Parameters**
- **callback** *(Function)* – the callback to delete.

**Raises**
- **ValueError** – if callback is not in the callback list of `digi.xbee.reader.SocketDataReceived` event.

**del_socket_data_received_from_callback** *(callback)*
 Deletes a callback for the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

**Parameters**
- **callback** *(Function)* – the callback to delete.

**Raises**
- **ValueError** – if callback is not in the callback list of `digi.xbee.reader.SocketDataReceivedFrom` event.

**del_socket_state_received_callback** *(callback)*
 Deletes a callback for the callback list of `digi.xbee.reader.SocketStateReceived` event.

**Parameters**
- **callback** *(Function)* – the callback to delete.

**Raises**
- **ValueError** – if callback is not in the callback list of `digi.xbee.reader.SocketStateReceived` event.

**del_user_data_relay_received_callback** *(callback)*
 Deletes a callback for the callback list of `digi.xbee.reader.RelayDataReceived` event.

**Parameters**
- **callback** *(Function)* – the callback to delete.

**Raises**
- **ValueError** – if callback is not in the callback list of `digi.xbee.reader.RelayDataReceived` event.

**disable_bluetooth** ()
 Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

**Raises**
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**enable_apply_changes** *(value)*
Sets the apply_changes flag.

**Parameters**
- **value** *(Boolean)* – True to enable the apply changes flag, False to disable it.

**enable_bluetooth** ()
 Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

**Raises**
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBe device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
execute_command(parameter)
  Executes the provided command.

   **Parameters** parameter (String) – The name of the AT command to be executed.

   **Raises**
   - **TimeoutException** – if the response is not received before the read timeout expires.
   - **XBeeException** – if the XBees device’s serial port is closed.
   - **InvalidOperatingModeException** – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
   - **ATCommandException** – if the response is not as expected.

flush_queues()
Flushes the packets queue.

get_16bit_addr()
Deprecated.

This protocol does not have an associated 16-bit address.

get_64bit_addr()
Returns the 64-bit address of the XBee device.

   **Returns** the 64-bit address of the XBee device.

   **Return type** XBee64BitAddress

See also:

XBee64BitAddress

get_adc_value(io_line)
Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.set_io_configuration() and IOMode.ADC.

   **Parameters** io_line (IOLine) – the IO line to get its ADC value.

   **Returns** the analog value corresponding to the provided IO line.

   **Return type** Integer

   **Raises**
   - **TimeoutException** – if the response is not received before the read timeout expires.
   - **XBeeException** – if the XBees device’s serial port is closed.
   - **InvalidOperatingModeException** – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
   - **ATCommandException** – if the response is not as expected.
   - **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

See also:
IOLine

get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

Returns the API output mode of the XBee device.

Return type APIOutputMode

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns the parameter value.

Return type Bytarray

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit
get_bluetooth_mac_addr()
Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

Returns  The Bluetooth MAC address.
return_type  String
Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

get_current_frame_id()
Returns the last used frame ID.

Returns  the last used frame ID.
Return_type  Integer

get_dest_address()
Deprecated.
Operation not supported in this protocol. Use IDevice.get_dest_ip_addr() instead. This method will raise an AttributeError.

get_dest_ip_addr()
Returns the destination IP address.

Returns  The configured destination IP address.
Return_type  ipaddress.IPv4Address
Raises

• TimeoutException – if there is a timeout getting the destination IP address.
• XBeeException – if there is any other XBee related exception.

See also:

ipaddress.IPv4Address

get_dio_value(io_line)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use AbstractXBeeDevice.set_io_configuration().

Parameters  io_line (IOLine) – the DIO line to gets its digital value.

Returns  current value of the provided IO line.
Return_type  IOValue
Raises

• TimeoutException – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

See also:

*IOLine*  
*IOValue*

### get_firmware_version()

Returns the firmware version of the XBee device.

**Returns**  
the hardware version of the XBee device.

**Return type**  
Bytearray

### get_hardware_version()

Returns the hardware version of the XBee device.

**Returns**  
the hardware version of the XBee device.

**Return type**  
*HardwareVersion*

See also:

*HardwareVersion*

### get_io_configuration(io_line)

Returns the configuration of the provided IO line.

**Parameters**  
**io_line** (*IOLine*) – the io line to configure.

**Returns**  
the IO mode of the IO line provided.

**Return type**  
*IOMode*

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the received data is not an IO mode.

### get_io_sampling_rate()

Returns the IO sampling rate of the XBee device.

**Returns**  
the IO sampling rate of XBee device.

**Return type**  
Integer
Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

`get_ip_addr()`

Returns the IP address of this IP device.

To refresh this value use the method `IPDevice.read_device_info()`.

- **Returns**: The IP address of this IP device.

- **Return type**: `ipaddress.IPv4Address`

See also:

`ipaddress.IPv4Address`

`get_network()`

Deprecated.

This protocol does not support the network functionality.

`get_next_frame_id()`

Returns the next frame ID of the XBee device.

- **Returns**: The next frame ID of the XBee device.

- **Return type**: `Integer`

`get_node_id()`

Returns the Node Identifier (NI) value of the XBee device.

- **Returns**: the Node Identifier (NI) of the XBee device.

- **Return type**: `String`

`get_pan_id()`

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

`get_parameter(param, parameter_value=None)`

Override.

- **See also:**
  
  `AbstractXBeeDevice.get_parameter()`

`get_power_level()`

Returns the power level of the XBee device.

- **Returns**: the power level of the XBee device.

- **Return type**: `PowerLevel`
Raises `TimeoutException` – if the response is not received before the read timeout expires.

See also:

_PowerLevel_

**get_pwm_duty_cycle**(io_line)

Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters `io_line` (**IOLine**) – the IO line to get its PWM duty cycle.

Returns the PWM duty cycle of the given IO line or `None` if the response is empty.

Return type `Integer`

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `ValueError` – if the passed IO_LINE has no PWM capability.

See also:

_IOLine_

**get_role**()

Gets the XBee role.

Returns the role of the XBee.

Return type `digi.xbee.models.protocol.Role`

See also:

`digi.xbee.models.protocol.Role`

**get_route_to_node**(remote, **timeout=10**)

Gets the route from this XBee to the given remote node.

Parameters

- `remote` (**RemoteXBeeDevice**) – The remote node.
- `timeout` (**Float, optional, default=10**) – Maximum number of seconds to wait for the route.

Returns
Tuple containing route data (None if the route was not read in the provided timeout):
- source (RemoteXBeeDevice): The source node of the route.
- destination (RemoteXBeeDevice): The destination node of the route.

- hops (List): List of intermediate nodes (RemoteXBeeDevice) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout() Returns the serial port read timeout.
    Returns the serial port read timeout in seconds.
    Return type Integer

get_xbee_device_callbacks() Returns this XBee internal callbacks for process received packets.
    This method is called by the PacketListener associated with this XBee to get its callbacks. These callbacks will be executed before user callbacks.
    Returns PacketReceived

has_explicit_packets() Returns whether the XBee device’s queue has explicit packets or not. This do not include non-explicit packets.
    Returns True if this XBee device’s queue has explicit packets, False otherwise.
    Return type Boolean

See also:
XBeeDevice.has_packets()

has_packets() Returns whether the XBee device’s queue has packets or not. This do not include explicit packets.
    Returns True if this XBee device’s queue has packets, False otherwise.
    Return type Boolean

See also:
XBeeDevice.has_explicit_packets()

is_apply_changes_enabled() Returns whether the apply_changes flag is enabled or not.
    Returns True if the apply_changes flag is enabled, False otherwise.
    Return type Boolean

is_open() Returns whether this XBee device is open or not.
Returns Boolean. True if this XBee device is open, False otherwise.

is_remote()
Override method.

See also:

AbstractXBeeDevice.is_remote()

log
Returns the XBee device log.

Returns the XBee device logger.

Return type Logger

operating_mode
Returns this XBee device’s operating mode.

Returns OperatingMode. This XBee device’s operating mode.

reachable
Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

read_data (timeout=None, explicit=False)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

read_data_from (remote_xbee_device, timeout=None, explicit=False)
Deprecated.

Operation not supported in this protocol. This method will raise an AttributeError.

read_device_info (init=True, fire_event=True)
Override.

See also:

AbstractXBeeDevice.read_device_info()

read_io_sample()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type IOSample

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ATCommandException` – if the response is not as expected.

See also:

`IOSample`

`read_ip_data(timeout=3)`
Reads new IP data received by this XBee device during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

Parameters `timeout` (Integer, optional) – The time to wait for new IP data in seconds.

Returns IP message, None if this device did not receive new data.

Return type `IPMessage`

Raises `ValueError` – if `timeout` is less than 0.

`read_ip_data_from(ip_addr, timeout=3)`
Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method `IPDevice.add_ip_data_received_callback()`.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method `IPDevice.start_listening()` for that purpose. When finished, you can use the method `IPDevice.stop_listening()` to stop listening for incoming IP data.

Parameters
• `ip_addr` (ipaddress.IPv4Address) – The IP address to read data from.
• `timeout` (Integer, optional) – The time to wait for new IP data in seconds.

Returns
IP message, None if this device did not receive new data from the provided IP address.

Return type `IPMessage`

Raises `ValueError` – if `timeout` is less than 0.

`reset()`
Override method.

See also:

`AbstractXBeeDevice.reset()`
scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data (data)
Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters data (Bytearray) – Data to send.

Raises

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• XBeeException – if there is any problem sending the data.

See also:

XBeedevic.send_micropython_data()
XBeedevic.send_user_data_relay()
send_ip_data_async(ip_addr, dest_port, protocol, data, close_socket=False)

Sends the provided IP data to the given IP address and port asynchronously using the specified IP protocol. For TCP and TCP SSL protocols, you can also indicate if the socket should be closed when data is sent.

Asynchronous transmissions do not wait for answer from the remote device or for transmit status packet.

Parameters

- **ip_addr** (ipaddress.IPv4Address) – The IP address to send IP data to.
- **dest_port** (Integer) – The destination port of the transmission.
- **protocol** (IPProtocol) – The IP protocol used for the transmission.
- **data** (String or bytearray) – The IP data to be sent.
- **close_socket** (Boolean, optional) – True to close the socket just after the transmission. False to keep it open. Default to False.

Raises

- **ValueError** – if **ip_addr** is None.
- **ValueError** – if **protocol** is None.
- **ValueError** – if **data** is None.
- **ValueError** – if **dest_port** is less than 0 or greater than 65535.
- **OperationNotSupportedException** – if the device is remote.
- **XBeeException** – if there is any other XBee related exception.

send_ip_data_broadcast(dest_port, data)

Sends the provided IP data to all clients.

This method blocks till a success or error transmit status arrives or the configured receive timeout expires.

Parameters

- **dest_port** (Integer) – The destination port of the transmission.
- **data** (String or bytearray) – The IP data to be sent.

Raises

- **ValueError** – if **data** is None.
- **ValueError** – if **dest_port** is less than 0 or greater than 65535.
- **TimeoutException** – if there is a timeout sending the data.
- **XBeeException** – if there is any other XBee related exception.

send_micropython_data(data)

Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters **data** (Bytearray) – Data to send.

Raises
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `XBeeException` – if there is any problem sending the data.

See also:

```
XBeeDevice.send_bluetooth_data()
XBeeDevice.send_user_data_relay()
```

**send_packet** *(packet, sync=False)*

Override method.

See also:

```
AbstractXBeeDevice._send_packet()
```

**send_packet_sync_and_get_response** *(packet_to_send, timeout=None)*

Override method.

See also:

```
AbstractXBeeDevice._send_packet_sync_and_get_response()
```

**send_user_data_relay** *(local_interface, data)*

Sends the given data to the given XBee local interface.

Parameters

• `local_interface` *(XBeeLocalInterface)* – Destination XBee local interface.
• `data` *(Bytearray)* – Data to send.

Raises

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ValueError` – if `local_interface` is `None`.
• `XBeeException` – if there is any problem sending the User Data Relay.

See also:

```
XBeeLocalInterface
```

**serial_port**

Returns the serial port associated to the XBee device, if any.

Returns

the serial port associated to the XBee device. Returns `None` if the local XBee does not use serial communication.
Return type: `XBeeSerialPort`

See also:

`XBeeSerialPort`

**set_16bit_addr** (*value*)

Sets the 16-bit address of the XBee device.

**Parameters**

- *value* (*XBee16BitAddress*) – the new 16-bit address of the XBee device.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the current protocol is 802.15.4.

**set_api_output_mode** (*api_output_mode*)

Deprecated since version 1.3: Use `set_api_output_mode_value()`

Sets the API output mode of the XBee device.

**Parameters**

- *api_output_mode* (*APIOutputMode*) – the new API output mode of the XBee device.

** Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the current protocol is ZigBee

See also:

`APIOutputMode`

**set_api_output_mode_value** (*api_output_mode*)

Sets the API output mode of the XBee.

**Parameters**

- *api_output_mode* (*Integer*) – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

**set_dest_address** *(addr)*

Deprecated.

Operation not supported in this protocol. Use *IPDevice.set_dest_ip_addr()* instead. This method will raise an *AttributeError*.

**set_dest_ip_addr** *(address)*

Sets the destination IP address.

Parameters *address* (ipaddress.IPv4Address) – Destination IP address.

Raises

• **ValueError** – if *address* is None.

• **TimeoutException** – if there is a timeout setting the destination IP address.

• **XBeeException** – if there is any other XBee related exception.

See also:

ipaddress.IPv4Address

**set_dio_change_detection** *(io_lines_set)*

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A None set of lines disables this feature.

Parameters *io_lines_set* – set of *IOLine*.

Raises

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

See also:

*IOLine*
set_dio_value(io_line, io_value)
Sets the digital value (high or low) to the provided IO line.

Parameters

- io_line (IOLine) – the digital IO line to sets its value.
- io_value (IOValue) – the IO value to set to the IO line.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

IOLine
IOValue

set_io_configuration(io_line, io_mode)
Sets the configuration of the provided IO line.

Parameters

- io_line (IOLine) – the IO line to configure.
- io_mode (IOMode) – the IO mode to set to the IO line.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

IOLine
IOMode

set_io_sampling_rate(rate)
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters rate (Integer) – the new IO sampling rate of the XBee device in seconds.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

`set_node_id(node_id)`
Sets the Node Identifier (NI) value of the XBee device.

Parameters
`node_id` *(String)* – the new Node Identifier (NI) of the XBee device.

Raises
• `ValueError` – if `node_id` is `None` or its length is greater than 20.

• `TimeoutException` – if the response is not received before the read timeout expires.

`set_pan_id(value)`
Deprecated.
Operation not supported in this protocol. This method will raise an `AttributeError`.

`set_parameter(param, value)`
Override.

See: `AbstractXBeeDevice.set_parameter()`

`set_power_level(power_level)`
Sets the power level of the XBee device.

Parameters
`power_level` *(PowerLevel)* – the new power level of the XBee device.

Raises
`TimeoutException` – if the response is not received before the read timeout expires.

See also:

`PowerLevel`

`set_pwm_duty_cycle(io_line, cycle)`
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

• `io_line` *(IOLine)* – the IO Line to be assigned.

• `cycle` *(Integer)* – duty cycle in % to be assigned. Must be between 0 and 100.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

• `ValueError` – if the given IO line does not have PWM capability or `cycle` is not between 0 and 100.

See also:
IOLine
IOMode.PWM

**set_sync_ops_timeout** *(sync_ops_timeout)*
Sets the serial port read timeout.

**Parameters**
- **sync_ops_timeout** *(Integer)* – the read timeout, expressed in seconds.

**start_listening** *(source_port)*
Starts listening for incoming IP transmissions in the provided port.

**Parameters**
- **source_port** *(Integer)* – Port to listen for incoming transmissions.

** Raises **
- **ValueError** – if **source_port** is less than 0 or greater than 65535.
- **TimeoutException** – if there is a timeout setting the source port.
- **XBeeException** – if there is any other XBee related exception.

**stop_listening** *( )*
Stops listening for incoming IP transmissions.

** Raises **
- **TimeoutException** – if there is a timeout processing the operation.
- **XBeeException** – if there is any other XBee related exception.

**update_bluetooth_password** *(new_password)*
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**
- **new_password** *(String)* – New Bluetooth password.

** Raises **
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from** *(device)*
Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**
- **device** *(AbstractXBeeDevice)* – the XBee device to get the data from.

**Returns**
- **True** if the device data has been updated, **False** otherwise.

**Return type**
Boolean

**update_firmware** *(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)*
Performs a firmware update operation of the device.

**Parameters**
- **xml_firmware_file** *(String)* – path of the XML file that describes the firmware to upload.
• **xbee_firmware_file** *(String, optional)* – location of the XBee binary firmware file.

• **bootloader_firmware_file** *(String, optional)* – location of the bootloader binary firmware file.

• **timeout** *(Integer, optional)* – the maximum time to wait for target read operations during the update process.

• **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  
  - The current update task as a String
  - The current update task percentage as an Integer

**Raises**

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.
- **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes**()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method `is_apply_configuration_changes_enabled()` to get its status and `enable_apply_configuration_changes()` to enable/disable the option. If it is disabled, method `apply_changes()` can be used in order to manually apply the changes.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**class** **RemoteXBeeDevice** *(local_xbee_device, x64bit_addr=<digi.xbee.models.address.XBee64BitAddress object>, x16bit_addr=<digi.xbee.models.address.XBee16BitAddress object>, node_id=None)*

**Bases:** **digi.xbee.devices.AbstractXBeeDevice**

This class represents a remote XBee device.

**Class constructor.** Instantiates a new **RemoteXBeeDevice** with the provided parameters.

**Parameters**
• **local_xbee_device** (*XBeeDevice*) – the local XBee device associated with the remote one.

• **x64bit_addr** (*XBee64BitAddress*) – the 64-bit address of the remote XBee device.

• **x16bit_addr** (*XBee16BitAddress*) – the 16-bit address of the remote XBee device.

• **node_id** (*String, optional*) – the node identifier of the remote XBee device. Optional.

See also:

XBee16BitAddress
XBee64BitAddress
XBeeDevice

get_parameter *(parameter, parameter_value=None)*

Override.

See also:

AbstractXBeeDevice.get_parameter()

set_parameter *(parameter, value)*

Override.

See also:

AbstractXBeeDevice.set_parameter()

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

reset()

Override method.

See also:

AbstractXBeeDevice.reset()

get_local_xbee_device()

Returns the local XBee device associated to the remote one.

Returns *XBeeDevice*
**set_local_xbee_device**(local_xbee_device)

This method associates a XBeeDevice to the remote XBee device.

**Parameters**

local_xbee_device (XBeeDevice) – the new local XBee device associated to the remote one.

**See also:**

XBeeDevice

---

**get_serial_port()**

Returns the serial port of the local XBee device associated to the remote one.

**Returns**

the serial port of the local XBee device associated to the remote one.

**Return type**

XBeeSerialPort

**See also:**

XBeeSerialPort

---

**get_comm_iface()**

Returns the communication interface of the local XBee device associated to the remote one.

**Returns**

the communication interface of the local XBee device associated to the remote one.

**Return type**

XBeeCommunicationInterface

**See also:**

XBeeCommunicationInterface

---

**apply_changes()**

Applies changes via AC command.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**apply_profile**(profile_path, progress_callback=None)

Applies the given XBee profile to the XBee device.

**Parameters**

- **profile_path**(String) – path of the XBee profile file to apply.
- **progress_callback**(Function, optional) –
function to execute to receive progress information. Receives two arguments:

- The current apply profile task as a String
- The current apply profile task percentage as an Integer

Raises

- XBeeException – if the device is not open.
- InvalidOperatingModeException – if the device operating mode is invalid.
- UpdateProfileException – if there is any error applying the XBee profile.
- OperationNotSupportedException – if XBee profiles are not supported in the XBee device.

disable_bluetooth()

Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

enable_apply_changes(value)

Sets the apply_changes flag.

Parameters value (Boolean) – True to enable the apply changes flag, False to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the AbstractXBeeDevice.update_bluetooth_password() method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

execute_command(parameter)

Executes the provided command.

Parameters parameter (String) – The name of the AT command to be executed.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• \textit{ATCommandException} – if the response is not as expected.

\textbf{get\_16bit\_addr()}

Returns the 16-bit address of the XBee device.

- **Returns** the 16-bit address of the XBee device.

- **Return type** \texttt{XBee16BitAddress}

See also:

- \texttt{XBee16BitAddress}

\textbf{get\_64bit\_addr()}

Returns the 64-bit address of the XBee device.

- **Returns** the 64-bit address of the XBee device.

- **Return type** \texttt{XBee64BitAddress}

See also:

- \texttt{XBee64BitAddress}

\textbf{get\_adc\_value} (\texttt{io\_line})

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use \texttt{AbstractXBeeDevice.set\_io\_configuration()} and \texttt{IOMode.ADC}.

- **Parameters** \texttt{io\_line (IOLine)} – the IO line to get its ADC value.

- **Returns** the analog value corresponding to the provided IO line.

- **Return type** Integer

- **Raises**

  - \textit{TimeoutException} – if the response is not received before the read timeout expires.

  - \textit{XBeeException} – if the XBee device’s serial port is closed.

  - \textit{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

  - \textit{ATCommandException} – if the response is not as expected.

  - \textit{OperationNotSupportedException} – if the response does not contain the value for the given IO line.

See also:

- \texttt{IOLine}

598 Chapter 2. Contents
get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

Returns the API output mode of the XBee device.

Return type APIOutputMode

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns the parameter value.

Return type bytearray

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.
Returns The Bluetooth MAC address.

Return type String

Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

`get_current_frame_id()`

Returns the last used frame ID.

Returns the last used frame ID.

Return type Integer

`get_dest_address()`

Returns the 64-bit address of the XBee device that data will be reported to.

Returns the address.

Return type `XBee64BitAddress`

Raises `TimeoutException` – if the response is not received before the read timeout expires.

See also:

`XBee64BitAddress`

`get_dio_value(io_line)`

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

Parameters `io_line` (`IOLine`) – the DIO line to get its digital value.

Returns current value of the provided IO line.

Return type `IOValue`

Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

`IOLine`
`IOValue`
**get_firmware_version()**

Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type: Bytearray

**get_hardware_version()**

Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type: HardwareVersion

See also:

*HardwareVersion*

**get_io_configuration(io_line)**

Returns the configuration of the provided IO line.

Parameters:

- **io_line** (IOLine) – the io line to configure.

Returns the IO mode of the IO line provided.

Return type: IOMode

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **OperationNotSupportedException** – if the received data is not an IO mode.

**get_io_sampling_rate()**

Returns the IO sampling rate of the XBee device.

Returns the IO sampling rate of XBee device.

Return type: Integer

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**get_node_id()**

Returns the Node Identifier (NI) value of the XBee device.

Returns the Node Identifier (NI) of the XBe device.

Return type: String
get_pan_id()
Returns the operating PAN ID of the XBee device.

Returns operating PAN ID of the XBee device.
Return type Bytearray
Raises TimeoutException – if the response is not received before the read timeout expires.

get_power_level()
Returns the power level of the XBee device.

Returns the power level of the XBee device.
Return type PowerLevel
Raises TimeoutException – if the response is not received before the read timeout expires.

See also:

PowerLevel

get_protocol()
Returns the current protocol of the XBee device.

Returns the current protocol of the XBee device.
Return type XBeeProtocol

See also:

XBeeProtocol

get_pwm_duty_cycle(io_line)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters io_line (IOLine) – the IO line to get its PWM duty cycle.

Returns the PWM duty cycle of the given IO line or None if the response is empty.
Return type Integer

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• ValueError – if the passed IO_LINE has no PWM capability.

See also:

IOLine
get_role()  
Gets the XBee role.

Returns the role of the XBee.

Return type `digi.xbee.models.protocol.Role`

See also:

`digi.xbee.models.protocol.Role`

get_sync_ops_timeout()  
Returns the serial port read timeout.

Returns the serial port read timeout in seconds.

Return type Integer

is_apply_changes_enabled()  
Returns whether the apply_changes flag is enabled or not.

Returns `True` if the apply_changes flag is enabled, `False` otherwise.

Return type Boolean

log  
Returns the XBee device log.

Returns the XBee device logger.

Return type Logger

reachable  
Returns whether the XBee is reachable.

Returns `True` if the device is reachable, `False` otherwise.

Return type Boolean

read_device_info(init=True, fire_event=True)  
Updates all instance parameters reading them from the XBee device.

Parameters

• `init` *(Boolean, optional, default=`True`) – If False only not initialized parameters are read, all if True.*

• `fire_event` *(Boolean, optional, default=`True`) – True to launch and update event if any parameter changed, False otherwise.*

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

read_io_sample()  
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.
Returns the IO sample read from the XBee device.

Return type **IOSample**

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

**IOSample**

**scan_counter**

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type **Integer**

**set_16bit_addr**(value)

Sets the 16-bit address of the XBee device.

Parameters value (**XBee16BitAddress**) – the new 16-bit address of the XBee device.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **OperationNotSupportedException** – if the current protocol is not 802.15.4.

**set_api_output_mode**(api_output_mode)

Deprecated since version 1.3: Use **set_api_output_mode_value()**

Sets the API output mode of the XBee device.

Parameters api_output_mode (**APIOutputMode**) – the new API output mode of the XBee device.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **OperationNotSupportedException** – if the current protocol is ZigBee
See also:

APIOutputMode

**set_api_output_mode_value** (*api_output_mode*)
Sets the API output mode of the XBee.

Parameters *api_output_mode* (*Integer*) – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBe device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.
- *OperationNotSupportedException* – if it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

**set_dest_address** (*addr*)
Sets the 64-bit address of the XBee device that data will be reported to.

Parameters *addr* (*XBee64BitAddress* or *RemoteXBeeDevice*) – the address itself or the remote XBee device that you want to set up its address as destination address.

Raises

- *TimeoutException* – If the response is not received before the read timeout expires.
- *XBeeException* – If the XBe device’s serial port is closed.
- *InvalidOperatingModeException* – If the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – If the response is not as expected.
- *ValueError* – If *addr* is *None*.

**set_dio_change_detection** (*io_lines_set*)
Sets the digital IO lines to be monitored and sampled whenever their status changes.

A *None* set of lines disables this feature.

Parameters *io_lines_set* – set of IOLine.

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

See also:

**IOLine**

### `set_dio_value(io_line, io_value)`
Sets the digital value (high or low) to the provided IO line.

**Parameters**

• **io_line (IOLine)** – the digital IO line to sets its value.

• **io_value (IOValue)** – the IO value to set to the IO line.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

See also:

**IOLine**

**IOValue**

### `set_io_configuration(io_line, io_mode)`
Sets the configuration of the provided IO line.

**Parameters**

• **io_line (IOLine)** – the IO line to configure.

• **io_mode (IOMode)** – the IO mode to set to the IO line.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

See also:

**IOLine**
IOMode

**set_io_sampling_rate**(rate)
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters `rate` (**Integer**) – the new IO sampling rate of the XBee device in seconds.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.
• `XBeeException` – if the XBees device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ATCommandException` – if the response is not as expected.

**set_node_id**(node_id)
Sets the Node Identifier (NI) value of the XBee device.

Parameters `node_id` (**String**) – the new Node Identifier (NI) of the XBee device.

Raises

• `ValueError` – if `node_id` is `None` or its length is greater than 20.
• `TimeoutException` – if the response is not received before the read timeout expires.

**set_pan_id**(value)
Sets the operating PAN ID of the XBee device.

Parameters `value` (**Bytearray**) – the new operating PAN ID of the XBee device. Must have only 1 or 2 bytes.

Raises `TimeoutException` – if the response is not received before the read timeout expires.

**set_power_level**(power_level)
Sets the power level of the XBee device.

Parameters `power_level` (**PowerLevel**) – the new power level of the XBee device.

Raises `TimeoutException` – if the response is not received before the read timeout expires.

See also: PowerLevel

**set_pwm_duty_cycle**(io_line, cycle)
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

• `io_line` (**IOLine**) – the IO Line to be assigned.
• `cycle` (**Integer**) – duty cycle in % to be assigned. Must be between 0 and 100.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• ValueError – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

IOLine
IOMode.PWM

set_sync_ops_timeout(sync_ops_timeout)
Sets the serial port read timeout.

Parameters sync_ops_timeout (Integer) – the read timeout, expressed in seconds.

update_bluetooth_password(new_password)
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

Parameters new_password (String) – New Bluetooth password.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

update_device_data_from(device)
Updates the current device reference with the data provided for the given device.

This is only for internal use.

Parameters device (AbstractXBeeDevice) – the XBee device to get the data from.

Returns True if the device data has been updated, False otherwise.

Return type Boolean

update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)
Performs a firmware update operation of the device.

Parameters
• xml_firmware_file (String) – path of the XML file that describes the firmware to upload.
• xbee_firmware_file (String, optional) – location of the XBee binary firmware file.
• bootloader_firmware_file (String, optional) – location of the bootloader binary firmware file.
• **timeout** (*Integer, optional*) – the maximum time to wait for target read operations during the update process.

• **progress_callback** (*Function, optional*) – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

**Raises**

• **XBeeException** – if the device is not open.

• **InvalidOperatingModeException** – if the device operating mode is invalid.

• **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.

• **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes()**

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method `is_apply_configuration_changes_enabled()` to get its status and `enable_apply_configuration_changes()` to enable/disable the option. If it is disabled, method `apply_changes()` can be used in order to manually apply the changes.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

**class** `digi.xbee.devices.RemoteRaw802Device` (*local_xbee_device*, *x64bit_addr=None*, *x16bit_addr=None*, *node_id=None*)

**Bases:** `digi.xbee.devices.RemoteXBeeDevice`

This class represents a remote 802.15.4 XBee device.

Class constructor. Instantiates a new `RemoteXBeeDevice` with the provided parameters.

**Parameters**

• **local_xbee_device** (*XBeeDevice*) – the local XBee device associated with the remote one.

• **x64bit_addr** (*XBee64BitAddress*) – the 64-bit address of the remote XBee device.

• **x16bit_addr** (*XBee16BitAddress*) – the 16-bit address of the remote XBee device.
• **node_id (String, optional)** – the node identifier of the remote XBee device. Optional.

  Raises **XBeeException** – if the protocol of `local_xbee_device` is invalid.

  See also:

  RemoteXBeeDevice
  XBee16BitAddress
  XBee64BitAddress
  XBeeDevice

  `get_protocol()`
  Override.

  See also:

  RemoteXBeeDevice.get_protocol()

  `set_64bit_addr(address)`
  Sets the 64-bit address of this remote 802.15.4 device.

  Parameters **address (XBee64BitAddress)** – The 64-bit address to be set to the device.

  Raises **ValueError** – if `address` is `None`.

  `get_ai_status()`
  Override.

  See also:

  AbstractXBeeDevice._get_ai_status()

  `apply_changes()`
  Applies changes via `AC` command.

  Raises

  • **TimeoutException** – if the response is not received before the read timeout expires.
  
  • **XBeeException** – if the XBee device’s serial port is closed.
  
  • **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  
  • **ATCommandException** – if the response is not as expected.

  `apply_profile(profile_path, progress_callback=None)`
  Applies the given XBee profile to the XBee device.

  Parameters

  • **profile_path (String)** – path of the XBee profile file to apply.
  
  • **progress_callback (Function, optional)** –
function to execute to receive progress information. Receives two arguments:

- The current apply profile task as a String
- The current apply profile task percentage as an Integer

Raises

- XBeeException – if the device is not open.
- InvalidOperatingModeException – if the device operating mode is invalid.
- UpdateProfileException – if there is any error applying the XBee profile.
- OperationNotSupportedException – if XBee profiles are not supported in the XBee device.

disable_bluetooth()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

enable_apply_changes(value)
Sets the apply_changes flag.

Parameters value (Boolean) – True to enable the apply changes flag, False to disable it.

enable_bluetooth()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the AbstractXBeeDevice.update_bluetooth_password() method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

execute_command(parameter)
Executes the provided command.

Parameters parameter (String) – The name of the AT command to be executed.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

**get_16bit_addr()**

Returns the 16-bit address of the XBee device.

**Returns**  the 16-bit address of the XBee device.

**Return type**  XBee16BitAddress

See also:

XBee16BitAddress

**get_64bit_addr()**

Returns the 64-bit address of the XBee device.

**Returns**  the 64-bit address of the XBee device.

**Return type**  XBee64BitAddress

See also:

XBee64BitAddress

**get_adc_value(io_line)**

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice. set_io_configuration() and IOMode.ADC.

**Parameters** **io_line** (**IOLine**) – the IO line to get its ADC value.

**Returns**  the analog value corresponding to the provided IO line.

**Return type**  Integer

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

See also:

IOLine
get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

Returns the API output mode of the XBee device.

Return type APIOutputMode

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

Returns the parameter value.

Return type Bytearray

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.
Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

`get_comm_iface()`

Returns the communication interface of the local XBee device associated to the remote one.

Returns

- the communication interface of the local XBee device associated to the remote one.

Return type `XBeeCommunicationInterface`

See also: `XBeeCommunicationInterface`

`get_current_frame_id()`

Returns the last used frame ID.

Returns

- the last used frame ID.

Return type `Integer`

`get_dest_address()`

Returns the 64-bit address of the XBee device that data will be reported to.

Returns

- the address.

Return type `XBee64BitAddress`

Raises `TimeoutException` – if the response is not received before the read timeout expires.

See also: `XBee64BitAddress`

`get_dio_value(io_line)`

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

Parameters

- `io_line` (IOLine) – the DIO line to gets its digital value.

Returns

- current value of the provided IO line.

Return type `IOValue`

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

See also:

`IOLine`  
`IOValue`

### get_firmware_version()

Returns the firmware version of the XBee device.

- **Returns** the hardware version of the XBee device.
- **Return type** Bytearray

### get_hardware_version()

Returns the hardware version of the XBee device.

- **Returns** the hardware version of the XBee device.
- **Return type** `HardwareVersion`

See also:

`HardwareVersion`

### get_io_configuration(io_line)

Returns the configuration of the provided IO line.

- **Parameters** `io_line` (*IOLine*) – the io line to configure.
- **Returns** the IO mode of the IO line provided.
- **Return type** `IOMode`

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **OperationNotSupportedException** – if the received data is not an IO mode.

### get_io_sampling_rate()

Returns the IO sampling rate of the XBee device.

- **Returns** the IO sampling rate of XBee device.
- **Return type** Integer
Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

### get_local_xbee_device()

Returns the local XBee device associated to the remote one.

**Returns**  
`XBeeDevice`

### get_node_id()

Returns the Node Identifier (NI) value of the XBee device.

**Returns**  
the Node Identifier (NI) of the XBee device.

**Return type**  
String

### get_pan_id()

Returns the operating PAN ID of the XBee device.

**Returns**  
operating PAN ID of the XBee device.

**Return type**  
Bytearray

**Raises**  
`TimeoutException` – if the response is not received before the read timeout expires.

### get_parameter

*override.*

**See also:**

`AbstractXBeeDevice.get_parameter()`

### get_power_level()

Returns the power level of the XBee device.

**Returns**  
the power level of the XBee device.

**Return type**  
`PowerLevel`

**Raises**  
`TimeoutException` – if the response is not received before the read timeout expires.

**See also:**

`PowerLevel`

### get_pwm_duty_cycle

*override.*

**Parameters**  
`io_line` (IOLine) – the IO line to get its PWM duty cycle.

**Returns**  
the PWM duty cycle of the given IO line or `None` if the response is empty.

**Return type**  
Integer
Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `ValueError` – if the passed `IO_LINE` has no PWM capability.

See also:

`IOLine`

get_role()

Gets the XBee role.

- **Returns** the role of the XBee.
- **Return type** `digi.xbee.models.protocol.Role`

See also:

`digi.xbee.models.protocol.Role`

get_serial_port()

Returns the serial port of the local XBee device associated to the remote one.

- **Returns** the serial port of the local XBee device associated to the remote one.
- **Return type** `XBeeSerialPort`

See also:

`XBeeSerialPort`

get_sync_ops_timeout()

Returns the serial port read timeout.

- **Returns** the serial port read timeout in seconds.
- **Return type** Integer

is_apply_changes_enabled()

Returns whether the apply_changes flag is enabled or not.

- **Returns** True if the apply_changes flag is enabled, False otherwise.
- **Return type** Boolean

is_remote()

Override method.

See also:
AbstractXBeeDevice.is_remote()

log
Returns the XBee device log.

Returns the XBee device logger.

Return type Logger

reachable
Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

read_device_info (init=True, fire_event=True)
Updates all instance parameters reading them from the XBee device.

Parameters

• init (Boolean, optional, default=True) – If False only not initialized
  parameters are read, all if True.

• fire_event (Boolean, optional, default=True) – True to launch and
  update event if any parameter changed, False otherwise.

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not
  API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

read_io_sample()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input
channels.

Returns the IO sample read from the XBee device.

Return type IOSample

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not
  API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

See also:

IOSample
reset()  
Override method.  
See also:

AbstractXBeeDevice.reset()

scan_counter  
Returns the scan counter for this node.  
Returns The scan counter for this node.  
Return type Integer

set_16bit_addr(value)  
Sets the 16-bit address of the XBee device.  
Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.  
Raises

• TimeoutException – if the response is not received before the read timeout expires.  
• XBeeException – if the XBee device’s serial port is closed.  
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.  
• ATCommandException – if the response is not as expected.  
• OperationNotSupportedException – if the current protocol is not 802.15.4.

set_api_output_mode(api_output_mode)  
Deprecated since version 1.3: Use set_api_output_mode_value()  
Sets the API output mode of the XBee device.  
Parameters api_output_mode (APIOutputMode) – the new API output mode of the XBee device.  
Raises

• TimeoutException – if the response is not received before the read timeout expires.  
• XBeeException – if the XBee device’s serial port is closed.  
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.  
• ATCommandException – if the response is not as expected.  
• OperationNotSupportedException – if the current protocol is ZigBee  
See also:  

APIOutputMode

set_api_output_mode_value(api_output_mode)  
Sets the API output mode of the XBee.
Parameters `api_output_mode` *(Integer)* – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

`set_dest_address` *(addr)*

Sets the 64-bit address of the XBee device that data will be reported to.

**Parameters** `addr` *(XBee64BitAddress or RemoteXBeeDevice)* – the address itself or the remote XBee device that you want to set up its address as destination address.

**Raises**

- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If the response is not as expected.
- `ValueError` – If `addr` is None.

`set_dio_change_detection` *(io_lines_set)*

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A None set of lines disables this feature.

**Parameters** `io_lines_set` – set of `IOLine`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

See also:
set_dio_value (io_line, io_value)
Sets the digital value (high or low) to the provided IO line.

Parameters
- io_line (IOLine) – the digital IO line to sets its value.
- io_value (IOValue) – the IO value to set to the IO line.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:
IOLine
IOValue

set_io_configuration (io_line, io_mode)
Sets the configuration of the provided IO line.

Parameters
- io_line (IOLine) – the IO line to configure.
- io_mode (IOMode) – the IO mode to set to the IO line.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:
IOLine
IOMode

set_io_sampling_rate (rate)
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters rate (Integer) – the new IO sampling rate of the XBee device in seconds.

Raises
• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

### set_local_xbee_device

`set_local_xbee_device(local_xbee_device)`

This method associates a `XBeeDevice` to the remote XBee device.

**Parameters** `local_xbee_device` (`XBeeDevice`) – the new local XBee device associated to the remote one.

**See also:**

`XBeeDevice`

### set_node_id

`set_node_id(node_id)`

Sets the Node Identifier (NI) value of the XBee device.

**Parameters** `node_id` (`String`) – the new Node Identifier (NI) of the XBee device.

**Raises**

• `ValueError` – if `node_id` is `None` or its length is greater than 20.

• `TimeoutException` – if the response is not received before the read timeout expires.

### set_pan_id

`set_pan_id(value)`

Sets the operating PAN ID of the XBee device.

**Parameters** `value` (`Bytearray`) – the new operating PAN ID of the XBee device. Must have only 1 or 2 bytes.

**Raises** `TimeoutException` – if the response is not received before the read timeout expires.

### set_parameter

`set_parameter(parameter, value)`

Override.

**See also:**

`AbstractXBeeDevice.set_parameter()`

### set_power_level

`set_power_level(power_level)`

Sets the power level of the XBee device.

**Parameters** `power_level` (`PowerLevel`) – the new power level of the XBee device.

**Raises** `TimeoutException` – if the response is not received before the read timeout expires.

**See also:**

`PowerLevel`
**set_pwm_duty_cycle**(*io_line, cycle*)

Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

**Parameters**

- *io_line (IOLine*) – the IO Line to be assigned.
- *cycle (Integer*) – duty cycle in % to be assigned. Must be between 0 and 100.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **ValueError** – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

**See also:**

IOLine
IOMode.PWM

**set_sync_ops_timeout**(*sync_ops_timeout*)

Sets the serial port read timeout.

**Parameters** *sync_ops_timeout (Integer*) – the read timeout, expressed in seconds.

**update_bluetooth_password**(*new_password*)

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters** *new_password (String*) – New Bluetooth password.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from**(*device*)

Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters** *device (AbstractXBeeDevice*) – the XBee device to get the data from.

**Returns** True if the device data has been updated, False otherwise.

**Return type** Boolean
update_firmware (xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)

Performs a firmware update operation of the device.

Parameters

• `xml_firmware_file` (String) – path of the XML file that describes the firmware to upload.

• `xbee_firmware_file` (String, optional) – location of the XBee binary firmware file.

• `bootloader_firmware_file` (String, optional) – location of the bootloader binary firmware file.

• `timeout` (Integer, optional) – the maximum time to wait for target read operations during the update process.

• `progress_callback` (Function, optional) – function to execute to receive progress information. Receives two arguments:

  – The current update task as a String
  – The current update task percentage as an Integer

Raises

• `XBeeException` – if the device is not open.

• `InvalidOperatingModeException` – if the device operating mode is invalid.

• `OperationNotSupportedException` – if the firmware update is not supported in the XBee device.

• `FirmwareUpdateException` – if there is any error performing the firmware update.

write_changes ()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method `is_apply_configuration_changes_enabled()` to get its status and `enable_apply_configuration_changes()` to enable/disable the option. If it is disabled, method `apply_changes()` can be used in order to manually apply the changes.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.
class digi.xbee.devices.RemoteDigiMeshDevice:
    __init__(local_xbee_device, x64bit_addr=None, node_id=None)

Bases: digi.xbee.devices.RemoteXBeeDevice

This class represents a remote DigiMesh XBee device.

Class constructor. Instantiates a new RemoteDigiMeshDevice with the provided parameters.

Parameters

- **local_xbee_device** (XBeeDevice) – the local XBee device associated with the remote one.
- **x64bit_addr** (XBee64BitAddress) – the 64-bit address of the remote XBee device.
- **node_id** (String, optional) – the node identifier of the remote XBee device. Optional.

Raises XBeeException – if the protocol of local_xbee_device is invalid.

See also: RemoteXBeeDevice

XBee64BitAddress

XBeeDevice

get_protocol()

Override.

See also:

RemoteXBeeDevice.get_protocol()

get_neighbors(neighbor_callback=None, process_finished_callback=None, timeout=None)

Returns the neighbors of this XBee. If neighbor_callback is not defined, the process blocks during the specified timeout.

Parameters

- **neighbor_callback** (Function, optional, default=``None``) – method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
- **process_finished_callback** (Function, optional, default=``None``) – method to execute when the process finishes. Receives two arguments:
  - The XBee device that is searching for its neighbors.
  - A list with the discovered neighbors.
  - An error message if something went wrong.
- **timeout** (Float, optional, default=\``NeighborFinder.DEFAULT_TIMEOUT``) – The timeout in seconds.
Returns

```
List of com.digi.models.zdo.Neighbor when neighbor_callback is defined. None otherwise (in this case neighbors are received in the callback).
```

Return type List

Raises OperationNotSupportedException – If XBee protocol is not DigiMesh.

See also:

com.digi.models.zdo.Neighbor

**apply_changes()**

Applies changes via AC command.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBeed device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeed device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**apply_profile** *(profile_path, progress_callback=None)*

Applies the given XBeed profile to the XBeed device.

Parameters

- **profile_path** *(String)* – path of the XBeed profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – if the device is not open.
- `InvalidOperatingModeException` – if the device operating mode is invalid.
- `UpdateProfileException` – if there is any error applying the XBeed profile.
- `OperationNotSupportedException` – if XBeed profiles are not supported in the XBeed device.

**disable_bluetooth()**

Disables the Bluetooth interface of this XBeed device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBeed device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**enable_apply_changes**(*value*)

Sets the apply_changes flag.

**Parameters**

- **value** (*Boolean*) – True to enable the apply changes flag, False to disable it.

**enable_bluetooth**()

Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the `AbstractXBeeDevice.update_bluetooth_password()` method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**execute_command**(*parameter*)

Executes the provided command.

**Parameters**

- **parameter** (*String*) – The name of the AT command to be executed.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**get_16bit_addr**()

Returns the 16-bit address of the XBee device.

**Returns**

- the 16-bit address of the XBee device.

**Return type** *XBee16BitAddress*

See also:

```
XBee16BitAddress
```

**get_64bit_addr**()

Returns the 64-bit address of the XBee device.

**Returns**

- the 64-bit address of the XBee device.

**Return type** *XBee64BitAddress*

See also:

```
XBee64BitAddress
```
get_adc_value(io_line)

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.
set_io_configuration() and IOMode.ADC.

Parameters  io_line (IOLine) – the IO line to get its ADC value.

Returns  the analog value corresponding to the provided IO line.

Return type  Integer

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBees device’s serial port is closed.
- InvalidOperatingModeException – if the XBees device’s operating mode is not
  API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the response does not contain the value
  for the given IO line.

See also:

IOLine

get_api_output_mode()

Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of
the XBee device.

Returns  the API output mode of the XBee device.

Return type  APIOutputMode

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBees device’s serial port is closed.
- InvalidOperatingModeException – if the XBees device’s operating mode is not
  API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.
The API output mode determines the format that the received data is output through the serial interface of the XBee.

**Returns** the parameter value.

**Return type** Bytearray

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

### get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

**Returns** The Bluetooth MAC address.

**Return type** String

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

### get_comm_iface()

Returns the communication interface of the local XBee device associated to the remote one.

**Returns**

**the communication interface of the local XBee device associated to** the remote one.

**Return type** XBeeCommunicationInterface

See also:

`XBeeCommunicationInterface`

### get_current_frame_id()

Returns the last used frame ID.

**Returns** the last used frame ID.
Return type Integer

**get_dest_address()**
Returns the 64-bit address of the XBee device that data will be reported to.

Returns the address.

Return type `XBee64BitAddress`

Raises `TimeoutException` – if the response is not received before the read timeout expires.

See also:

`XBee64BitAddress`

**get_dio_value**(io_line)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

Parameters **io_line** (`IOLine`) – the DIO line to get its digital value.

Returns current value of the provided IO line.

Return type `IOValue`

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

`IOLine`
`IOValue`

**get_firmware_version()**
Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type `Bytearray`

**get_hardware_version()**
Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type `HardwareVersion`

See also:
get_io_configuration (io_line)
Returns the configuration of the provided IO line.

Parameters io_line (IOLine) – the io line to configure.

Returns the IO mode of the IO line provided.

Return type IOMode

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBe device’s serial port is closed.
• InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• OperationNotSupportedException – if the received data is not an IO mode.

get_io_sampling_rate()
Returns the IO sampling rate of the XBee device.

Returns the IO sampling rate of XBee device.

Return type Integer

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBe device’s serial port is closed.
• InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

get_local_xbee_device()
Returns the local XBe device associated to the remote one.

Returns XBeeDevice

get_node_id()
Returns the Node Identifier (NI) value of the XBe device.

Returns the Node Identifier (NI) of the XBe device.

Return type String

get_pan_id()
Returns the operating PAN ID of the XBe device.

Returns operating PAN ID of the XBe device.

Return type bytearray

Raises TimeoutException – if the response is not received before the read timeout expires.

get_parameter (parameter, parameter_value=None)
Override.

See also:
AbstractXBeeDevice.get_parameter()

get_power_level()

Returns the power level of the XBee device.

    Returns  the power level of the XBee device.

    Return type  PowerLevel

    Raises  TimeoutException – if the response is not received before the read timeout expires.

See also:

PowerLevel

get_pwm_duty_cycle (io_line)

Returns the PWM duty cycle in % corresponding to the provided IO line.

    Parameters  io_line (IOLine) – the IO line to get its PWM duty cycle.

    Returns  the PWM duty cycle of the given IO line or None if the response is empty.

    Return type  Integer

    Raises

    •  TimeoutException – if the response is not received before the read timeout expires.

    •  XBeeException – if the XBee device’s serial port is closed.

    •  InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

    •  ATCommandException – if the response is not as expected.

    •  ValueError – if the passed IO_LINE has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

    Returns  the role of the XBee.

    Return type  digi.xbee.models.protocol.Role

See also:

digi.xbee.models.protocol.Role

get_serial_port()

Returns the serial port of the local XBee device associated to the remote one.

    Returns  the serial port of the local XBee device associated to the remote one.
Return type: XBeeSerialPort

See also:

XBeeSerialPort

get_sync_ops_timeout() — Returns the serial port read timeout.

Returns the serial port read timeout in seconds.

Return type: Integer

is_apply_changes_enabled() — Returns whether the apply_changes flag is enabled or not.

Returns True if the apply_changes flag is enabled, False otherwise.

Return type: Boolean

is_remote() — Override method.

See also:

AbstractXBeeDevice.is_remote()

log — Returns the XBee device log.

Returns the XBee device logger.

Return type: Logger

reachable — Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type: Boolean

read_device_info(init=True, fire_event=True) — Updates all instance parameters reading them from the XBee device.

Parameters

- init (Boolean, optional, default='True') – If False only not initialized parameters are read, all if True.
- fire_event (Boolean, optional, default='True') – True to launch and update event if any parameter changed, False otherwise.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
read_io_sample()  
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type: IOSample

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

IOSample

reset()  
Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter  
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type: Integer

set_16bit_addr(value)  
Sets the 16-bit address of the XBee device.

Parameters

value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the current protocol is not 802.15.4.

set_api_output_mode(api_output_mode)  
Deprecated since version 1.3: Use set_api_output_mode_value()

Sets the API output mode of the XBee device.
Parameters `api_output_mode` (**APIOutputMode**) – the new API output mode of the XBee device.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

**See also:**

**APIOutputMode**

`set_api_output_mode_value(api_output_mode)`

Sets the API output mode of the XBee.

**Parameters** `api_output_mode` (**Integer**) – new API output mode options. Calculate this value using the method `digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value()` with a set of `digi.xbee.models.mode.APIOutputModeBit`.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if it is not supported by the current protocol.

**See also:**

`digi.xbee.models.mode.APIOutputModeBit`

`set_dest_address(addr)`

Sets the 64-bit address of the XBee device that data will be reported to.

**Parameters** `addr` (**XBee64BitAddress** or **RemoteXBeeDevice**) – the address itself or the remote XBee device that you want to set up its address as destination address.

**Raises**

- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• `ATCommandException` – If the response is not as expected.

• `ValueError` – If `addr` is `None`.

`set_dio_change_detection`(*io_lines_set*)

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A `None` set of lines disables this feature.

**Parameters**

`io_lines_set` – set of `IOLine`.

**Raises**

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

See also:

`IOLine`

`set_dio_value`(*io_line*, *io_value*)

Sets the digital value (high or low) to the provided IO line.

**Parameters**

• `io_line` (*IOLine*) – the digital IO line to sets its value.

• `io_value` (*IOValue*) – the IO value to set to the IO line.

**Raises**

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

See also:

`IOLine`

`IOValue`

`set_io_configuration`(*io_line*, *io_mode*)

Sets the configuration of the provided IO line.

**Parameters**

• `io_line` (*IOLine*) – the IO line to configure.

• `io_mode` (*IOMode*) – the IO mode to set to the IO line.

**Raises**

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBe device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**set_io_sampling_rate***(rate)**
Sets the IO sampling rate of the XBe device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

**Parameters** *

- **rate***(Integer)** – the new IO sampling rate of the XBe device in seconds.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBe device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**set_local_xbee_device***(local_xbee_device)**
This method associates a XBe device to the remote XBe device.

**Parameters** *

- **local_xbee_device***(XBeDevice)** – the new local XBe device associated to the remote one.

**See also:**

**XBeeDevice**

**set_node_id***(node_id)**
Sets the Node Identifier (NI) value of the XBe device.

**Parameters** *

- **node_id***(String)** – the new Node Identifier (NI) of the XBe device.

**Raises**

- **ValueError** – if **node_id** is **None** or its length is greater than 20.
- **TimeoutException** – if the response is not received before the read timeout expires.

**set_pan_id***(value)**
Sets the operating PAN ID of the XBe device.

**Parameters** *

- **value***(Bytearray)** – the new operating PAN ID of the XBe device. Must have only 1 or 2 bytes.

**Raises** **TimeoutException** – if the response is not received before the read timeout expires.
set_parameter parameter, value

Override.

See also:

AbstractXBeeDevice.set_parameter()

set_power_level power_level

Sets the power level of the XBee device.

Parameters power_level (PowerLevel) – the new power level of the XBee device.

Raises TimeoutException – if the response is not received before the read timeout expires.

See also:

PowerLevel

set_pwm_duty_cycle io_line, cycle

Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

• io_line (IOLine) – the IO Line to be assigned.
• cycle (Integer) – duty cycle in % to be assigned. Must be between 0 and 100.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• ValueError – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

IOLine
IOMode.PWM

set_sync_ops_timeout sync_ops_timeout

Sets the serial port read timeout.

Parameters sync_ops_timeout (Integer) – the read timeout, expressed in seconds.

update_bluetooth_password new_password

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.
Parameters **new_password** *(String)* – New Bluetooth password.

Raise

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from** *(device)*

Updates the current device reference with the data provided for the given device.

This is only for internal use.

Parameters **device** *(AbstractXBeeDevice)* – the XBee device to get the data from.

Returns True if the device data has been updated, False otherwise.

Return type Boolean

**update_firmware** *(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)*

Performs a firmware update operation of the device.

Parameters

- **xml_firmware_file** *(String)* – path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** *(String, optional)* – location of the XBee binary firmware file.
- **bootloader_firmware_file** *(String, optional)* – location of the bootloader binary firmware file.
- **timeout** *(Integer, optional)* – the maximum time to wait for target read operations during the update process.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raise

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **OperationNotSupportedException** – if the firmware update is not supported in the XBee device.
- **FirmwareUpdateException** – if there is any error performing the firmware update.

**write_changes** ()

Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.
Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method `is_apply_configuration_changes_enabled()` to get its status and `enable_apply_configuration_changes()` to enable/disable the option. If it is disabled, method `apply_changes()` can be used in order to manually apply the changes.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

```python
class digi.xbee.devices.RemoteDigiPointDevice(local_xbee_device, x64bit_addr=None, node_id=None)
```

Bases: `digi.xbee.devices.RemoteXBeeDevice`

This class represents a remote DigiPoint XBee device.

Class constructor. Instantiates a new `RemoteDigiMeshDevice` with the provided parameters.

**Parameters**

- `local_xbee_device (XBeeDevice)` – the local XBee device associated with the remote one.
- `x64bit_addr (XBee64BitAddress)` – the 64-bit address of the remote XBee device.
- `node_id (String, optional)` – the node identifier of the remote XBee device. Optional.

**Raises** `XBeeException` – if the protocol of `local_xbee_device` is invalid.

See also:

- `RemoteXBeeDevice`
- `XBee64BitAddress`
- `XBeeDevice`

```python
get_protocol()
```

Override.

**See also:**

- `RemoteXBeeDevice.get_protocol()`

```python
apply_changes()
```

Applies changes via AC command.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
apply_profile(profile_path, progress_callback=None)
Applies the given XBee profile to the XBee device.

Parameters
• profile_path(String) – path of the XBee profile file to apply.
• progress_callback(Function, optional) – 
  function to execute to receive progress information. Receives two arguments:
  – The current apply profile task as a String
  – The current apply profile task percentage as an Integer

Raises
• XBeeException – if the device is not open.
• InvalidOperatingModeException – if the device operating mode is invalid.
• UpdateProfileException – if there is any error applying the XBee profile.
• OperationNotSupportedException – if XBee profiles are not supported in the XBee device.

disable_bluetooth()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBe device’s serial port is closed.
• InvalidOperatingModeException – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

enable_apply_changes(value)
Sets the apply_changes flag.

Parameters value(Boolean) – True to enable the apply changes flag, False to disable it.

enable_bluetooth()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the AbstractXBeeDevice.update_bluetooth_password() method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
**execute_command** *(parameter)*

Executes the provided command.

**Parameters**

*parameter* *(String)* – The name of the AT command to be executed.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

**get_16bit_addr()**

Returns the 16-bit address of the XBe device.

**Returns** the 16-bit address of the XBe device.

**Return type** `XBee16BitAddress`

**See also:** `XBee16BitAddress`

**get_64bit_addr()**

Returns the 64-bit address of the XBe device.

**Returns** the 64-bit address of the XBe device.

**Return type** `XBee64BitAddress`

**See also:** `XBee64BitAddress`

**get_adc_value** *(io_line)*

Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use `AbstractXBeeDevice.set_io_configuration()` and `IOMode.ADC`.

**Parameters**

*io_line* *(IOLine)* – the IO line to get its ADC value.

**Returns** the analog value corresponding to the provided IO line.

**Return type** `Integer`

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBe device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
• **OperationNotSupportedException** – if the response does not contain the value for the given IO line.

See also:

*IOLine*

**get_api_output_mode()**

Deprecated since version 1.3: Use `get_api_output_mode_value()`

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

**Returns** the API output mode of the XBee device.

**Return type** `APIOutputMode`

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

See also:

`APIOutputMode`

**get_api_output_mode_value()**

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

**Returns** the parameter value.

**Return type** `Bytearray`

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.
• **XBeeException** – if the XBee device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.
• **OperationNotSupportedException** – if it is not supported by the current protocol.

See also:
**get_bluetooth_mac_addr()**
Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

- **Returns** The Bluetooth MAC address.
- **Return type** String
- **Raises**
  - `TimeoutException` – if the response is not received before the read timeout expires.
  - `XBeeException` – if the XBee device’s serial port is closed.
  - `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**get_comm_iface()**
Returns the communication interface of the local XBee device associated to the remote one.

- **Returns**
  - the communication interface of the local XBee device associated to the remote one.
- **Return type** `XBeeCommunicationInterface`
- **See also:**
  - `XBeeCommunicationInterface`

**get_current_frame_id()**
Returns the last used frame ID.

- **Returns** the last used frame ID.
- **Return type** Integer

**get_dest_address()**
Returns the 64-bit address of the XBee device that data will be reported to.

- **Returns** the address.
- **Return type** `XBee64BitAddress`
- **Raises** `TimeoutException` – if the response is not received before the read timeout expires.

- **See also:**
  - `XBee64BitAddress`

**get_dio_value(io_line)**
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use `AbstractXBeeDevice.set_io_configuration()`.

**digi.xbee.models.mode.APIOutputModeBit**
Parameters `io_line` (*IOLine*) – the DIO line to get its digital value.

Returns current value of the provided IO line.

Return type *IOValue*

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `OperationNotSupportedException` – if the response does not contain the value for the given IO line.

See also:

- `IOLine`
- `IOValue`

---

**get_firmware_version()**

Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type *Bytearray*

---

**get_hardware_version()**

Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type *HardwareVersion*

See also:

- `HardwareVersion`

---

**get_io_configuration(io_line)**

Returns the configuration of the provided IO line.

Parameters `io_line` (*IOLine*) – the IO line to configure.

Returns the IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• \texttt{ATCommandException} – if the response is not as expected.

• \texttt{OperationNotSupportedException} – if the received data is not an IO mode.

\textbf{get\_io\_sampling\_rate()}

Returns the IO sampling rate of the XBee device.

\textbf{Returns} the IO sampling rate of XBee device.

\textbf{Return type} Integer

\textbf{Raises}

• \texttt{TimeoutException} – if the response is not received before the read timeout expires.

• \texttt{XBeeException} – if the XBee device’s serial port is closed.

• \texttt{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• \texttt{ATCommandException} – if the response is not as expected.

\textbf{get\_local\_xbee\_device()}

Returns the local XBee device associated to the remote one.

\textbf{Returns} XBeeDevice

\textbf{get\_node\_id()}

Returns the Node Identifier (NI) value of the XBee device.

\textbf{Returns} the Node Identifier (NI) of the XBee device.

\textbf{Return type} String

\textbf{get\_pan\_id()}

Returns the operating PAN ID of the XBee device.

\textbf{Returns} operating PAN ID of the XBee device.

\textbf{Return type} Bytearray

\textbf{Raises} \texttt{TimeoutException} – if the response is not received before the read timeout expires.

\textbf{get\_parameter} (\texttt{parameter, parameter\_value=\texttt{None}})

Override.

\textbf{See also}:

\begin{verbatim}
  AbstractXBeeDevice.get\_parameter()
\end{verbatim}

\textbf{get\_power\_level()}

Returns the power level of the XBee device.

\textbf{Returns} the power level of the XBee device.

\textbf{Return type} \texttt{PowerLevel}

\textbf{Raises} \texttt{TimeoutException} – if the response is not received before the read timeout expires.

\textbf{See also}:

\begin{verbatim}
  PowerLevel
\end{verbatim}
get_pwm_duty_cycle(io_line)

Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters

io_line (IOLine) – the IO line to get its PWM duty cycle.

Returns

the PWM duty cycle of the given IO line or None if the response is empty.

Return type

Integer

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- ValueError – if the passed IO_LINE has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns

the role of the XBee.

Return type
digi.xbee.models.protocol.Role

See also:

digi.xbee.models.protocol.Role

get_serial_port()

Returns the serial port of the local XBee device associated to the remote one.

Returns

the serial port of the local XBee device associated to the remote one.

Return type

XBeeSerialPort

See also:

XBeeSerialPort

get_sync_ops_timeout()

Returns the serial port read timeout.

Returns

the serial port read timeout in seconds.

Return type

Integer

is_apply_changes_enabled()

Returns whether the apply_changes flag is enabled or not.
Returns True if the apply_changes flag is enabled, False otherwise.

Return type Boolean

is_remote()
Override method.

See also:

AbstractXBeeDevice.is_remote()

log
Returns the XBee device log.

Returns the XBee device logger.

Return type Logger

reachable
Returns whether the XBee is reachable.

Returns True if the device is reachable, False otherwise.

Return type Boolean

read_device_info(init=True, fire_event=True)
Updates all instance parameters reading them from the XBee device.

Parameters

• init (Boolean, optional, default=`True`) – If False only not initialized parameters are read, all if True.

• fire_event (Boolean, optional, default=`True`) – True to launch and update event if any parameter changed, False otherwise.

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

read_io_sample()
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type IOSample

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

See also:

*IOSample*

**reset()**

Override method.

See also:

*AbstractXBeeDevice.reset()*

**scan_counter**

Returns the scan counter for this node.

**Returns**  
The scan counter for this node.

**Return type**  
Integer

**set_16bit_addr(value)**

Sets the 16-bit address of the XBee device.

**Parameters**  
**value** (*XBee16BitAddress*) – the new 16-bit address of the XBee device.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the current protocol is not 802.15.4.

**set_api_output_mode(api_output_mode)**

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee device.

**Parameters**  
**api_output_mode** (*APIOutputMode*) – the new API output mode of the XBee device.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **OperationNotSupportedException** – if the current protocol is ZigBee
APIOutputMode

set_api_output_mode_value(api_output_mode)
Sets the API output mode of the XBee.

Parameters:
api_output_mode (Integer) – new API output mode options. Calculate this value using the method digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value() with a set of digi.xbee.models.mode.APIOutputModeBit.

Raises:
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

set_dest_address(addr)
Sets the 64-bit address of the XBee device that data will be reported to.

Parameters:
addr (XBee64BitAddress or RemoteXBeeDevice) – the address itself or the remote XBee device that you want to set up its address as destination address.

Raises:
- TimeoutException – If the response is not received before the read timeout expires.
- XBeeException – If the XBee device’s serial port is closed.
- InvalidOperatingModeException – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If the response is not as expected.
- ValueError – If addr is None.

set_dio_change_detection(io_lines_set)
Sets the digital IO lines to be monitored and sampled whenever their status changes.
A None set of lines disables this feature.

Parameters:
io_lines_set – set of IOLine.

 Raises:
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

See also:

*IOLine*

**set_dio_value**(*io_line*, *io_value*)

Sets the digital value (high or low) to the provided IO line.

Parameters

• *io_line* (**IOLine**) – the digital IO line to sets its value.

• *io_value* (**IOValue**) – the IO value to set to the IO line.

Raises

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

See also:

*IOLine*  
*IOValue*

**set_io_configuration**(*io_line*, *io_mode*)

Sets the configuration of the provided IO line.

Parameters

• *io_line* (**IOLine**) – the IO line to configure.

• *io_mode* (**IOMode**) – the IO mode to set to the IO line.

Raises

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

See also:

*IOLine*  
*IOMode*
**set_io_sampling_rate** *(rate)*

Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

**Parameters**

- **rate** *(Integer)* – the new IO sampling rate of the XBee device in seconds.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**set_local_xbee_device** *(local_xbee_device)*

This methods associates a `XBeeDevice` to the remote XBee device.

**Parameters**

- **local_xbee_device** *(XBeeDevice)* – the new local XBee device associated to the remote one.

**See also:**

`XBeeDevice`

**set_node_id** *(node_id)*

Sets the Node Identifier (NI) value of the XBee device.

**Parameters**

- **node_id** *(String)* – the new Node Identifier (NI) of the XBee device.

**Raises**

- **ValueError** – if `node_id` is `None` or its length is greater than 20.
- **TimeoutException** – if the response is not received before the read timeout expires.

**set_pan_id** *(value)*

Sets the operating PAN ID of the XBee device.

**Parameters**

- **value** *(Bytearray)* – the new operating PAN ID of the XBee device. Must have only 1 or 2 bytes.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

**set_parameter** *(parameter, value)*

Override.

**See also:**

`AbstractXBeeDevice.set_parameter()`

**set_power_level** *(power_level)*

Sets the power level of the XBee device.

**Parameters**

- **power_level** *(PowerLevel)* – the new power level of the XBee device.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
See also:

PowerLevel

**set_pwm_duty_cycle**(io_line, cycle)

Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

**Parameters**

- io_line *(IOLine)* – the IO Line to be assigned.
- cycle *(Integer)* – duty cycle in % to be assigned. Must be between 0 and 100.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
- `ValueError` – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

IOLine
IOMode.PWM

**set_sync_ops_timeout**(sync_ops_timeout)

Sets the serial port read timeout.

**Parameters**

- sync_ops_timeout *(Integer)* – the read timeout, expressed in seconds.

**update_bluetooth_password**(new_password)

Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**

- new_password *(String)* – New Bluetooth password.

**Raises**

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

**update_device_data_from**(device)

Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**

- device *(AbstractXBeeDevice)* – the XBee device to get the data from.
Returns True if the device data has been updated, False otherwise.

Return type Boolean

update_firmware (xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)
Performs a firmware update operation of the device.

Parameters

• xml_firmware_file (String) – path of the XML file that describes the firmware to upload.
• xbee_firmware_file (String, optional) – location of the XBee binary firmware file.
• bootloader_firmware_file (String, optional) – location of the bootloader binary firmware file.
• timeout (Integer, optional) – the maximum time to wait for target read operations during the update process.
• progress_callback (Function, optional) – function to execute to receive progress information. Receives two arguments:
  – The current update task as a String
  – The current update task percentage as an Integer

Raises

• XBeeException – if the device is not open.
• InvalidOperatingModeException – if the device operating mode is invalid.
• OperationNotSupportedException – if the firmware update is not supported in the XBee device.
• FirmwareUpdateException – if there is any error performing the firmware update.

write_changes ()
Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.

Parameters values remain in this device’s memory until overwritten by subsequent use of this method.

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method is_apply_configuration_changes_enabled() to get its status and enable_apply_configuration_changes() to enable/disable the option. If it is disabled, method apply_changes() can be used in order to manually apply the changes.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
class digi.xbee.devices.RemoteZigBeeDevice(local_xbee_device, x64bit_addr=None, x16bit_addr=None, node_id=None)  
Bases: digi.xbee.devices.RemoteXBeeDevice

This class represents a remote ZigBee XBee device.

Class constructor. Instantiates a new RemoteDigiMeshDevice with the provided parameters.

Parameters

- local_xbee_device (XBeeDevice) – the local XBee device associated with the remote one.
- x64bit_addr (XBee64BitAddress) – the 64-bit address of the remote XBee device.
- x16bit_addr (XBee16BitAddress) – the 16-bit address of the remote XBee device.
- node_id (String, optional) – the node identifier of the remote XBee device. Optional.

Raises XBeeException – if the protocol of local_xbee_device is invalid.

See also:

RemoteXBeeDevice
XBee16BitAddress
XBee64BitAddress
XBeeDevice

parent

Returns the parent of the XBee if it is an end device.

Returns

The parent of the node for end devices, None if unknown or if it is not an end device.

Return type AbstractXBeeDevice

get_protocol()  
Override.

See also:

RemoteXBeeDevice.get_protocol()

get_ai_status()  
Override.

See also:

AbstractXBeeDevice._get_ai_status()

force_disassociate()  
Override.

See also:
AbstractXBeeDevice._force_disassociate()

**get_routes (route_callback=None, process_finished_callback=None, timeout=None)**

Returns the routes of this XBee. If `route_callback` is not defined, the process blocks until the complete routing table is read.

**Parameters**

- **route_callback** *(Function, optional, default=`None`)* — method called when a new route is received. Receives two arguments:
  - The XBee that owns this new route.
  - The new route.

- **process_finished_callback** *(Function, optional, default=`None`)* — method to execute when the process finishes. Receives two arguments:
  - The XBee device that executed the ZDO command.
  - A list with the discovered routes.
  - An error message if something went wrong.

- **timeout** *(Float, optional, default=`RouteTableReader.DEFAULT_TIMEOUT`)* — The ZDO command timeout in seconds.

**Returns**

List of `com.digi.models.zdo.Route` when `route_callback` is defined, `None` otherwise (in this case routes are received in the callback).

**Return type** List

**Raises** `OperationNotSupportedException` — If XBee protocol is not Zigbee or Smart Energy.

**See also:**

`com.digi.models.zdo.Route`

**get_neighbors (neighbor_callback=None, process_finished_callback=None, timeout=None)**

Returns the neighbors of this XBee. If `neighbor_callback` is not defined, the process blocks until the complete neighbor table is read.

**Parameters**

- **neighbor_callback** *(Function, optional, default=`None`)* — method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.

- **process_finished_callback** *(Function, optional, default=`None`)* — method to execute when the process finishes. Receives two arguments:
  - The XBee device that executed the ZDO command.
- A list with the discovered neighbors.
- An error message if something went wrong.

- **timeout** *(Float, optional, default=``NeighborTableReader.DEFAULT_TIMEOUT``)* – The ZDO command timeout in seconds.

Returns

List of **com.digi.models.zdo.Neighbor** when **neighbor_callback** is defined, **None** otherwise (in this case neighbors are received in the callback).

Return type  List

Raises **OperationNotSupportedException** – If XBee protocol is not Zigbee or Smart Energy.

See also:

`com.digi.models.zdo.Neighbor`

### apply_changes()
Applies changes via AC command.

_raises_

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBe device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBe device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

### apply_profile(profile_path, progress_callback=None)
Applies the given XBee profile to the XBe device.

Parameters

- **profile_path** *(String)* – path of the XBee profile file to apply.
- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- **XBeeException** – if the device is not open.
- **InvalidOperatingModeException** – if the device operating mode is invalid.
- **UpdateProfileException** – if there is any error applying the XBee profile.
- **OperationNotSupportedException** – if XBee profiles are not supported in the XBe device.
disable_bluetooth()
Disables the Bluetooth interface of this XBee device.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

enable_apply_changes(value)
Sets the apply_changes flag.

Parameters value (Boolean) – True to enable the apply changes flag, False to disable it.

enable_bluetooth()
Enables the Bluetooth interface of this XBee device.

To work with this interface, you must also configure the Bluetooth password if not done previously. You can use the *AbstractXBeeDevice.update_bluetooth_password()* method for that purpose.

Note that your device must have Bluetooth Low Energy support to use this method.

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

execute_command(parameter)
Executes the provided command.

Parameters parameter (String) – The name of the AT command to be executed.

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.

get_16bit_addr()
Returns the 16-bit address of the XBee device.

Returns the 16-bit address of the XBee device.

Return type XBee16BitAddress

See also:

*XBee16BitAddress*
**get_64bit_addr()**
Returns the 64-bit address of the XBee device.

- **Returns**: the 64-bit address of the XBee device.
- **Return type**: XBee64BitAddress

See also:

XBee64BitAddress

**get_adc_value(io_line)**
Returns the analog value of the provided IO line.

The provided IO line must be previously configured as ADC. To do so, use AbstractXBeeDevice.set_io_configuration() and IOMode.ADC.

- **Parameters**: io_line (IOLine) – the IO line to get its ADC value.
- **Returns**: the analog value corresponding to the provided IO line.
- **Return type**: Integer
- **Raises**
  - TimeoutException – if the response is not received before the read timeout expires.
  - XBeeException – if the XBee device’s serial port is closed.
  - InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  - ATCommandException – if the response is not as expected.
  - OperationNotSupportedException – if the response does not contain the value for the given IO line.

See also:

IOLine

**get_api_output_mode()**
Deprecated since version 1.3: Use get_api_output_mode_value()

Returns the API output mode of the XBee device.

The API output mode determines the format that the received data is output through the serial interface of the XBee device.

- **Returns**: the API output mode of the XBee device.
- **Return type**: APIOutputMode
- **Raises**
  - TimeoutException – if the response is not received before the read timeout expires.
  - XBeeException – if the XBee device’s serial port is closed.
  - InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• \textit{ATCommandException} – if the response is not as expected.

See also:

\texttt{APIOutputMode}

\texttt{get\_api\_output\_mode\_value()}

Returns the API output mode of the XBee.

The API output mode determines the format that the received data is output through the serial interface of the XBee.

\textbf{Returns} the parameter value.

\textbf{Return type} Bytearray

\textbf{Raises}

• \textit{TimeoutException} – if the response is not received before the read timeout expires.

• \textit{XBeeException} – if the XBee device’s serial port is closed.

• \textit{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• \textit{ATCommandException} – if the response is not as expected.

• \textit{OperationNotSupportedException} – if it is not supported by the current protocol.

See also:

\texttt{digi.xbee.models.mode.APIOutputModeBit}

\texttt{get\_bluetooth\_mac\_addr()}

Reads and returns the EUI-48 Bluetooth MAC address of this XBee device in a format such as 00112233AABB.

Note that your device must have Bluetooth Low Energy support to use this method.

\textbf{Returns} The Bluetooth MAC address.

\textbf{Return type} String

\textbf{Raises}

• \textit{TimeoutException} – if the response is not received before the read timeout expires.

• \textit{XBeeException} – if the XBee device’s serial port is closed.

• \textit{InvalidOperatingModeException} – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

\texttt{get\_comm\_iface()}

Returns the communication interface of the local XBee device associated to the remote one.

\textbf{Returns}

the communication interface of the local XBee device associated to the remote one.

\textbf{Return type} XBeeCommunicationInterface
See also:

XBeeCommunicationInterface

get_current_frame_id()
Returns the last used frame ID.

Returns the last used frame ID.

Return type Integer

get_dest_address()
Returns the 64-bit address of the XBee device that data will be reported to.

Returns the address.

Return type XBee64BitAddress

Raises TimeoutException – if the response is not received before the read timeout expires.

See also:

XBee64BitAddress

get_dio_value(io_line)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use AbstractXBeeDevice.set_io_configuration().

Parameters io_line (IOLine) – the DIO line to gets its digital value.

Returns current value of the provided IO line.

Return type IOValue

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

• OperationNotSupportedException – if the response does not contain the value for the given IO line.

See also:

IOLine
IOValue

get_firmware_version()
Returns the firmware version of the XBee device.
Returns the hardware version of the XBee device.

Return type Bytearray

get_hardware_version()

Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type HardwareVersion

See also: HardwareVersion

get_io_configuration(io_line)

Returns the configuration of the provided IO line.

Parameters io_line (IOLine) – the io line to configure.

Returns the IO mode of the IO line provided.

Return type IOMode

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

• OperationNotSupportedException – if the received data is not an IO mode.

get_io_sampling_rate()

Returns the IO sampling rate of the XBee device.

Returns the IO sampling rate of XBee device.

Return type Integer

Raises

• TimeoutException – if the response is not received before the read timeout expires.

• XBeeException – if the XBee device’s serial port is closed.

• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• ATCommandException – if the response is not as expected.

get_local_xbee_device()

Returns the local XBee device associated to the remote one.

Returns XBeeDevice

get_node_id()

Returns the Node Identifier (NI) value of the XBee device.

Returns the Node Identifier (NI) of the XBee device.
Return type String

get_pan_id()  
Returns the operating PAN ID of the XBee device.

Returns operating PAN ID of the XBee device.

Return type bytearray

Raises TimeoutException – if the response is not received before the read timeout expires.

get_parameter (parameter, parameter_value=None)  
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()  
Returns the power level of the XBee device.

Returns the power level of the XBee device.

Return type PowerLevel

Raises TimeoutException – if the response is not received before the read timeout expires.

See also:

PowerLevel

get_pwm_duty_cycle (io_line)  
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters io_line (IOLine) – the IO line to get its PWM duty cycle.

Returns the PWM duty cycle of the given IO line or None if the response is empty.

Return type Integer

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.
• ValueError – if the passed IO_LINE has no PWM capability.

See also:

IOLine
get_role()
  Gets the XBee role.
  
  Returns  the role of the XBee.
  
  Return type  digi.xbee.models.protocol.Role

See also:

digi.xbee.models.protocol.Role

get_serial_port()
  Returns the serial port of the local XBee device associated to the remote one.
  
  Returns  the serial port of the local XBee device associated to the remote one.
  
  Return type  XBeeSerialPort

See also:

XBeeSerialPort

get_sync_ops_timeout()
  Returns the serial port read timeout.
  
  Returns  the serial port read timeout in seconds.
  
  Return type  Integer

is_apply_changes_enabled()
  Returns whether the apply_changes flag is enabled or not.
  
  Returns  True if the apply_changes flag is enabled, False otherwise.
  
  Return type  Boolean

is_remote()
  Override method.
  
  See also:

AbstractXBeeDevice.is_remote()

log
  Returns the XBee device log.
  
  Returns  the XBee device logger.
  
  Return type  Logger

reachable
  Returns whether the XBee is reachable.
  
  Returns  True if the device is reachable, False otherwise.
  
  Return type  Boolean
**read_device_info**(*init=True, fire_event=True*)
Updates all instance parameters reading them from the XBee device.

Parameters

- **init** *(Boolean, optional, default=`True`)* – If False only not initialized parameters are read, all if True.
- **fire_event** *(Boolean, optional, default=`True`)* – True to launch and update event if any parameter changed, False otherwise.

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

**read_io_sample()**
Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

Returns the IO sample read from the XBee device.

Return type **IOSample**

Raises

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

**IOSample**

**reset()**
Override method.

See also:

**AbstractXBeeDevice.reset()**

**scan_counter**
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type **Integer**
set_16bit_addr(value)
Sets the 16-bit address of the XBee device.

Parameters value (XBee16BitAddress) – the new 16-bit address of the XBee device.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the current protocol is not 802.15.4.

set_api_output_mode(api_output_mode)
Deprecated since version 1.3: Use set_api_output_mode_value()
Sets the API output mode of the XBee device.

Parameters api_output_mode (APIOutputMode) – the new API output mode of the XBee device.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if the current protocol is ZigBee

See also:

APIOutputMode

set_api_output_mode_value(api_output_mode)
Sets the API output mode of the XBee.

Parameters api_output_mode (Integer) – new API output mode options. Calculate this value using the method digi.xbee.models.mode.APIOutputModeBit.calculate_api_output_mode_value() with a set of digi.xbee.models.mode.APIOutputModeBit.

Raises
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.
- OperationNotSupportedException – if it is not supported by the current protocol.
See also:

digi.xbee.models.mode.APIOutputModeBit

**set_dest_address**(*addr*)

Sets the 64-bit address of the XBee device that data will be reported to.

Parameters *addr* (XBee64BitAddress or RemoteXBeeDevice) – the address itself or the remote XBee device that you want to set up its address as destination address.

Raises

- *TimeoutException* – If the response is not received before the read timeout expires.
- *XBeeException* – If the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – If the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – If the response is not as expected.
- *ValueError* – If *addr* is None.

**set_dio_change_detection**(*io_lines_set*)

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A None set of lines disables this feature.

Parameters *io_lines_set* – set of IOLine.

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *ATCommandException* – if the response is not as expected.

See also:

**set_dio_value**(*io_line*, *io_value*)

Sets the digital value (high or low) to the provided IO line.

Parameters

- *io_line* (IOLine) – the digital IO line to sets its value.
- *io_value* (IOValue) – the IO value to set to the IO line.

Raises

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

See also:

IOLine
IOValue

`set_io_configuration(io_line, io_mode)`
Sets the configuration of the provided IO line.

Parameters

• `io_line` (`IOLine`) – the IO line to configure.

• `io_mode` (`IOMode`) – the IO mode to set to the IO line.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

See also:

IOLine
IOMode

`set_io_sampling_rate(rate)`
Sets the IO sampling rate of the XBee device in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters `rate` (`Integer`) – the new IO sampling rate of the XBee device in seconds.

Raises

• `TimeoutException` – if the response is not received before the read timeout expires.

• `XBeeException` – if the XBee device’s serial port is closed.

• `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• `ATCommandException` – if the response is not as expected.

`set_local_xbee_device(local_xbee_device)`
This methods associates a `XBeeDevice` to the remote XBee device.

Parameters `local_xbee_device` (`XBeeDevice`) – the new local XBee device associated to the remote one.

See also:
set_node_id(node_id)
Sets the Node Identifier (NI) value of the XBee device.

Parameters

node_id (String) – the new Node Identifier (NI) of the XBee device.

Raises

• ValueError – if node_id is None or its length is greater than 20.
• TimeoutException – if the response is not received before the read timeout expires.

set_pan_id(value)
Sets the operating PAN ID of the XBee device.

Parameters

value (Bytearray) – the new operating PAN ID of the XBee device. Must have only 1 or 2 bytes.

Raises

TimeoutException – if the response is not received before the read timeout expires.

set_parameter(parameter, value)
Override.

See also:

AbstractXBeeDevice.set_parameter()

set_power_level(power_level)
Sets the power level of the XBee device.

Parameters

power_level (PowerLevel) – the new power level of the XBee device.

Raises

TimeoutException – if the response is not received before the read timeout expires.

See also:

PowerLevel

set_pwm_duty_cycle(io_line, cycle)
Sets the duty cycle in % of the provided IO line.

The provided IO line must be PWM-capable, previously configured as PWM output.

Parameters

• io_line (IOLine) – the IO Line to be assigned.
• cycle (Integer) – duty cycle in % to be assigned. Must be between 0 and 100.

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• **ATCommandException** – if the response is not as expected.

• **ValueError** – if the given IO line does not have PWM capability or cycle is not between 0 and 100.

See also:

```
IOLine
IOMode.PWM
```

### `set_sync_ops_timeout(sync_ops_timeout)`
Sets the serial port read timeout.

**Parameters**

- **sync_ops_timeout** *(Integer)* – the read timeout, expressed in seconds.

### `update_bluetooth_password(new_password)`
Changes the password of this Bluetooth device with the new one provided.

Note that your device must have Bluetooth Low Energy support to use this method.

**Parameters**

- **new_password** *(String)* – New Bluetooth password.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.

- **XBeeException** – if the XBee device’s serial port is closed.

- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

### `update_device_data_from(device)`
Updates the current device reference with the data provided for the given device.

This is only for internal use.

**Parameters**

- **device** *(AbstractXBeeDevice)* – the XBee device to get the data from.

**Returns**

- **True** if the device data has been updated, **False** otherwise.

**Return type**

Boolean

### `update_firmware(xml_firmware_file, xbee_firmware_file=None, bootloader_firmware_file=None, timeout=None, progress_callback=None)`
Performs a firmware update operation of the device.

**Parameters**

- **xml_firmware_file** *(String)* – path of the XML file that describes the firmware to upload.

- **xbee_firmware_file** *(String, optional)* – location of the XBee binary firmware file.

- **bootloader_firmware_file** *(String, optional)* – location of the bootloader binary firmware file.

- **timeout** *(Integer, optional)* – the maximum time to wait for target read operations during the update process.

- **progress_callback** *(Function, optional)* – function to execute to receive progress information. Receives two arguments:
– The current update task as a String
– The current update task percentage as an Integer

Raises
• XBeeException – if the device is not open.
• InvalidOperatingModeException – if the device operating mode is invalid.
• OperationNotSupportedException – if the firmware update is not supported in the XBee device.
• FirmwareUpdateException – if there is any error performing the firmware update.

write_changes()
Writes configurable parameter values to the non-volatile memory of the XBee device so that parameter modifications persist through subsequent resets.
Parameters values remain in this device’s memory until overwritten by subsequent use of this method.
If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.
Writing the parameter modifications does not mean those values are immediately applied, this depends on the status of the ‘apply configuration changes’ option. Use method is_apply_configuration_changes_enabled() to get its status and enable_apply_configuration_changes() to enable/disable the option. If it is disabled, method apply_changes() can be used in order to manually apply the changes.

Raises
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

class digi.xbee.devices.XBeeNetwork(xbee_device)
    Bases: object
This class represents an XBee Network.
The network allows the discovery of remote devices in the same network as the local one and stores them.
Class constructor. Instantiates a new XBeeNetwork.
Parameters xbee_device (XBeeDevice) – the local XBee device to get the network from.
Raises ValueError – if xbee_device is None.
ND_PACKET_FINISH = 1
Flag that indicates a “discovery process finish” packet.
ND_PACKET_REMOTE = 2
Flag that indicates a discovery process packet with info about a remote XBee device.
DEFAULT_TIME_BETWEEN_SCANS = 10
Default time (in seconds) to wait before starting a new scan.
MIN_TIME_BETWEEN_SCANS = 0
Low limit for the time (in seconds) to wait before starting a new scan.
MAX_TIME_BETWEEN_SCANS = 300
    High limit for the time (in seconds) to wait before starting a new scan.

DEFAULT_TIME_BETWEEN_REQUESTS = 5
    Default time (in seconds) to wait between node neighbors requests.

MIN_TIME_BETWEEN_REQUESTS = 0
    Low limit for the time (in seconds) to wait between node neighbors requests.

MAX_TIME_BETWEEN_REQUESTS = 300
    High limit for the time (in seconds) to wait between node neighbors requests.

SCAN_TIL_CANCEL = 0
    The neighbor discovery process continues until is manually stopped.

scan_counter
    Returns the scan counter.
    Returns The scan counter.
    Return type Integer

start_discovery_process (deep=False, n_deep_scans=1)
    Starts the discovery process. This method is not blocking.
    This process can discover node neighbors and connections, or only nodes:
    • Deep discovery: Network nodes and connections between them (including quality) are discovered.
      The discovery process will be running the number of scans configured in n_deep_scans. A scan is
      considered the process of discovering the full network. If there are more than one number of scans
      configured, after finishing one another is started, until n_deep_scans is satisfied.
      See set_deep_discovery_options() to establish the way the network discovery process is
      performed.
    • No deep discovery: Only network nodes are discovered.
      The discovery process will be running until the configured timeout expires or, in case of 802.15.4,
      until the ‘end’ packet is read.
      It may be that, after the timeout expires, there are devices that continue sending discovery packets to
      this XBee device. In this case, these devices will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or
their quality. The difference is the possibility of running more than one scan using a deep discovery.

Parameters
  • deep (Boolean, optional, default='False') - True for a deep network
    scan, looking for neighbors and their connections. False otherwise.
  • n_deep_scans (Integer, optional, default=1) - Number of scans to per- 
    form before automatically stopping the discovery process. SCAN_TIL_CANCEL 
    means the process will not be automatically stopped. Only applicable if deep=True.

See also:

XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_device_discovered_callback()
XBeeNetwork.del_discovery_process_finished_callback()
stop_discovery_process()
    Stops the discovery process if it is running.

    Note that DigiMesh/DigiPoint devices are blocked until the discovery time configured (NT parameter) has elapsed, so if you try to get/set any parameter during the discovery process you will receive a timeout exception.

discover_device(node_id)
    Blocking method. Discovers and reports the first remote XBee device that matches the supplied identifier.

    Parameters
    node_id (String) – the node identifier of the device to be discovered.

    Returns
    the discovered remote XBee device with the given identifier, None if the timeout expires and the device was not found.

    Return type RemoteXBeeDevice

discover_devices(device_id_list)
    Blocking method. Attempts to discover a list of devices and add them to the current network.

    This method does not guarantee that all devices of device_id_list will be found, even if they exist physically. This will depend on the node discovery operation (ND) and timeout.

    Parameters
    device_id_list (List) – list of device IDs to discover.

    Returns
    a list with the discovered devices. It may not contain all devices specified in device_id_list

    Return type List

is_discovery_running()
    Returns whether the discovery process is running or not.

    Returns
    True if the discovery process is running, False otherwise.

    Return type Boolean

get_devices()
    Returns a copy of the XBee devices list of the network.

    If another XBee device is added to the list before the execution of this method, this XBee device will not be added to the list returned by this method.

    Returns
    a copy of the XBee devices list of the network.

    Return type List

has_devices()
    Returns whether there is any device in the network or not.

    Returns
    True if there is at least one device in the network, False otherwise.

    Return type Boolean

get_number_devices()
    Returns the number of devices in the network.

    Returns
    the number of devices in the network.

    Return type Integer
add_network_modified_callback (callback)
Add a callback for the event digi.xbee.reader.NetworkModified.

Parameters callback (Function) – the callback. Receives one argument.
- The event type as a NetworkEventType
- The reason of the event as a NetworkEventReason
- The node added, updated or removed from the network as a XBeeDevice or RemoteXBeeDevice.

See also:

XBeeNetwork.del_network_modified_callback()

add_device_discovered_callback (callback)
Add a callback for the event DeviceDiscovered.

Parameters callback (Function) – the callback. Receives one argument.
- The discovered remote XBee device as a RemoteXBeeDevice

See also:

XBeeNetwork.del_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_discovery_process_finished_callback()

add_init_discovery_scan_callback (callback)
Add a callback for the event InitDiscoveryScan.

Parameters callback (Function) – the callback. Receives two arguments.
- Number of scan to start (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_init_discovery_scan_callback()

add_end_discovery_scan_callback (callback)
Add a callback for the event EndDiscoveryScan.

Parameters callback (Function) – the callback. Receives two arguments.
- Number of scan that has finished (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_end_discovery_scan_callback()
add_discovery_process_finished_callback (callback)
    Adds a callback for the event DiscoveryProcessFinished.

    Parameters callback (Function) – the callback. Receives one argument.
        • The event code as an Integer

    See also:

XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

del_network_modified_callback (callback)
    Deletes a callback for the callback list of digi.xbee.reader.NetworkModified.

    Parameters callback (Function) – the callback to delete.

    See also:

XBeeNetwork.add_network_modified_callback()

del_device_discovered_callback (callback)
    Deletes a callback for the callback list of DeviceDiscovered event.

    Parameters callback (Function) – the callback to delete.

    Raises ValueError – if callback is not in the callback list of DeviceDiscovered event.

    See also:

XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_discovery_process_finished_callback()

del_init_discovery_scan_callback (callback)
    Deletes a callback for the callback list of InitDiscoveryScan.

    Raises ValueError – if callback is not in the callback list of InitDiscoveryScan event.

    See also:

XBeeNetwork.add_init_discovery_scan_callback()

del_end_discovery_scan_callback (callback)
    Deletes a callback for the callback list of EndDiscoveryScan.

    Raises ValueError – if callback is not in the callback list of EndDiscoveryScan event.
See also:

XBeeNetwork.add_end_discovery_scan_callback()

del_discovery_process_finished_callback (callback)
Deletes a callback for the callback list of DiscoveryProcessFinished event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of DiscoveryProcessFinished event.

See also:

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

clear ()
Removes all the remote XBee devices from the network.

get_discovery_options ()
Returns the network discovery process options.

Returns the parameter value.

Return type Bytearray

Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

set_discovery_options (options)
Configures the discovery options (NO parameter) with the given value.

Parameters options (Set of DiscoveryOptions) – new discovery options, empty set to clear the options.

Raises

• ValueError – if options is None.
• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
• InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
• ATCommandException – if the response is not as expected.

See also:
DiscoveryOptions

get_deep_discovery_options()

Returns the deep discovery process options.

Returns

(digi.xbee.models.mode.NeighborDiscoveryMode, Boolean): Tuple containing:

• mode (digi.xbee.models.mode.NeighborDiscoveryMode): Neighbor discovery mode, the way to perform the network discovery process.

• remove_nodes (Boolean): True to remove nodes from the network if they were not discovered in the last scan, False otherwise.

Return type Tuple

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_deep_discovery_options(deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>, del_not_discovered_nodes_in_last_scan=False)

Configures the deep discovery options with the given values. These options are only applicable for “deep” discovery (see start_discovery_process())

Parameters

• deep_mode (NeighborDiscoveryMode, optional, default='NeighborDiscoveryMode.CASCADE') – Neighbor discovery mode, the way to perform the network discovery process.

• del_not_discovered_nodes_in_last_scan (Boolean, optional, default='False') – True to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_discovery_timeout()

Returns the network discovery timeout.

Returns the network discovery timeout.

Return type Float

Raises

2.5. API reference 677
• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

**set_discovery_timeout** *(discovery_timeout)*

Sets the discovery network timeout.

**Parameters**

**discovery_timeout** *(Float)* – timeout in seconds.

**Raises**

• **TimeoutException** – if the response is not received before the read timeout expires.

• **XBeeException** – if the XBee device’s serial port is closed.

• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **ValueError** – if `discovery_timeout` is not between the allowed minimum and maximum values.

**get_deep_discovery_timeouts()**

Gets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see `start_discovery_process()`).

**Returns**

Tuple containing:

• **node_timeout (Float): Maximum duration in seconds of the discovery process per node.**
  This used to find a node neighbors. This timeout is highly dependent on the nature of the network:
  – It should be greater than the highest NT (Node Discovery Timeout) of your network
  – and include enough time to let the message propagate depending on the sleep cycle of your devices.

• **time_bw_nodes (Float): Time to wait between node neighbors requests.** Use this setting not to saturate your network:
  – For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  – For ‘Flood’ the minimum time to wait between each node’s neighbor requests.

• **time_bw_scans (Float): Time to wait before starting a new network scan.**

**Return type** Tuple (Float, Float, Float)

**See also:**

`XBeeNetwork.set_deep_discovery_timeouts()`
XBeeNetwork.start_discovery_process()

**set_deep_discovery_timeouts**

Sets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see `start_discovery_process()`)

- **node_timeout** (Float, optional, default=None): Maximum duration in seconds of the discovery process used to find neighbors of a node. If None already configured timeouts are used.
- **time_bw_requests** (Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS'): Time to wait between node neighbors requests. It must be between `MIN_TIME_BETWEEN_REQUESTS` and `MAX_TIME_BETWEEN_REQUESTS` seconds inclusive. Use this setting not to saturate your network:
  - For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For ‘Flood’ the minimum time to wait between each node’s neighbor requests.
- **time_bw_scans** (Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS'): Time to wait before starting a new network scan. It must be between `MIN_TIME_BETWEEN_SCANS` and `MAX_TIME_BETWEEN_SCANS` seconds inclusive.

**Raises** ValueError – if node_timeout, time_bw_requests or time_bw_scans are not between their corresponding limits.

See also:

- XBeeNetwork.get_deep_discovery_timeouts()
- XBeeNetwork.start_discovery_process()

**get_device_by_64**

Returns the XBee in the network whose 64-bit address matches the given one.

- **Parameters** x64bit_addr (XBee64BitAddress) – The 64-bit address of the device to be retrieved.
- **Returns** the XBee device in the network or None if it is not found.
- **Return type** AbstractXBeeDevice
- **Raises** ValueError – if x64bit_addr is None or unknown.

**get_device_by_16**

Returns the XBee in the network whose 16-bit address matches the given one.

- **Parameters** x16bit_addr (XBee16BitAddress) – The 16-bit address of the device to be retrieved.
- **Returns** the XBee device in the network or None if it is not found.
- **Return type** AbstractXBeeDevice
- **Raises** ValueError – if x16bit_addr is None or unknown.

**get_device_by_node_id**

Returns the XBee in the network whose node identifier matches the given one.
 XBee Python Library Documentation, Release 1.3.0

Parameters node_id(String) – The node identifier of the device to be retrieved.

Returns the XBee device in the network or None if it is not found.

Return type AbstractXBeeDevice

Raises ValueError – if node_id is None.

add_if_not_exist(x64bit_addr=None, x16bit_addr=None, node_id=None)

Adds an XBee device with the provided parameters if it does not exist in the current network.

If the XBee device already exists, its data will be updated with the provided parameters that are not None.

Parameters

• x64bit_addr(XBee64BitAddress, optional) – XBee device’s 64bit address. Optional.

• x16bit_addr(XBee16BitAddress, optional) – XBee device’s 16bit address. Optional.

• node_id(String, optional) – the node identifier of the XBee device. Optional.

Returns

the remote XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns the given XBee device without changes.

Return type AbstractXBeeDevice

add.remote(remote_xbee_device)

Adds the provided remote XBee device to the network if it is not contained yet.

If the XBee device is already contained in the network, its data will be updated with the parameters of the XBee device that are not None.

Parameters remote_xbee_device(RemoteXBeeDevice) – the remote XBee device to add to the network.

Returns

the provided XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns it without changes.

Return type RemoteXBeeDevice

add_remotes(remote_xbee_devices)

Adds a list of remote XBee devices to the network.

If any XBee device of the list is already contained in the network, its data will be updated with the parameters of the XBee device that are not None.

Parameters remote_xbee_devices(List) – the list of RemoteXBeeDevice to add to the network.

remove_device(remote_xbee_device)

Removes the provided remote XBee device from the network.

Parameters remote_xbee_device(RemoteXBeeDevice) – the remote XBee device to be removed from the list.

Raises ValueError – if the provided RemoteXBeeDevice is not in the network.
get_discovery_callbacks()
Returns the API callbacks that are used in the device discovery process.
This callbacks notify the user callbacks for each XBee device discovered.

Returns

callback for generic devices discovery process, callback for discovery specific XBee
device ops.

Return type Tuple (Function, Function)

get_connections()
Returns a copy of the XBee connections.
If a new connection is added to the list after the execution of this method, this connection is not added to
the list returned by this method.

Returns A copy of the list of Connection for the network.

Return type List

get_node_connections(node)
Returns the network connections with one of their ends node.
If a new connection is added to the list after the execution of this method, this connection is not added to
the list returned by this method.

Returns List of Connection with node end.

Return type List

class digi.xbee.devices.ZigBeeNetwork(device)
Bases: digi.xbee.devices.XBeeNetwork
This class represents a ZigBee network.
The network allows the discovery of remote devices in the same network as the local one and stores them.
Class constructor. Instantiates a new ZigBeeNetwork.

Parameters device (ZigBeeDevice) – the local ZigBee device to get the network from.

Raises ValueError – if device is None.

add_device_discovered_callback(callback)
Adds a callback for the event DeviceDiscovered.

Parameters callback (Function) – the callback. Receives one argument.

• The discovered remote XBee device as a RemoteXBeeDevice

See also:

XBeeNetwork.del_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_discovery_process_finished_callback()

add_discovery_process_finished_callback(callback)
Adds a callback for the event DiscoveryProcessFinished.

Parameters callback (Function) – the callback. Receives one argument.

• The event code as an Integer
See also:

XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

add_end_discovery_scan_callback(callback)
Adds a callback for the event EndDiscoveryScan.

Parameters callback (Function) – the callback. Receives two arguments.
• Number of scan that has finished (starting with 1).
• Total number of scans.

See also:

XBeeNetwork.del_end_discovery_scan_callback()

add_if_not_exist(x64bit_addr=None, x16bit_addr=None, node_id=None)
Adds an XBee device with the provided parameters if it does not exist in the current network.
If the XBee device already exists, its data will be updated with the provided parameters that are not None.

Parameters
• x64bit_addr (XBee64BitAddress, optional) – XBee device’s 64bit address. Optional.
• x16bit_addr (XBee16BitAddress, optional) – XBee device’s 16bit address. Optional.
• node_id (String, optional) – the node identifier of the XBee device. Optional.

Returns
the remote XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns the given XBee device without changes.

Return type AbstractXBeeDevice

add_init_discovery_scan_callback(callback)
Adds a callback for the event InitDiscoveryScan.

Parameters callback (Function) – the callback. Receives two arguments.
• Number of scan to start (starting with 1).
• Total number of scans.

See also:

XBeeNetwork.del_init_discovery_scan_callback()
add_network_modified_callback(callback)
 Adds a callback for the event `digi.xbee.reader.NetworkModified`.

 **Parameters**
`callback (Function)` – the callback. Receives one argument.

• The event type as a `NetworkEventType`
• The reason of the event as a `NetworkEventReason`
• The node added, updated or removed from the network as a `XBeeDevice` or `RemoteXBeeDevice`.

See also:

`XBeeNetwork.del_network_modified_callback()`

add_remote(remote_xbee_device)
 Adds the provided remote XBee device to the network if it is not contained yet.

If the XBee device is already contained in the network, its data will be updated with the parameters of the XBee device that are not `None`.

**Parameters**
`remote_xbee_device (RemoteXBeeDevice)` – the remote XBee device to add to the network.

**Returns**
the provided XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns it without changes.

**Return type** `RemoteXBeeDevice`

add_remotes(remote_xbee_devices)
 Adds a list of remote XBee devices to the network.

If any XBee device of the list is already contained in the network, its data will be updated with the parameters of the XBee device that are not `None`.

**Parameters**
`remote_xbee_devices (List)` – the list of `RemoteXBeeDevice` to add to the network.

clear()
 Removes all the remote XBee devices from the network.

del_device_discovered_callback(callback)
 Deletes a callback for the callback list of `DeviceDiscovered` event.

**Parameters**
`callback (Function)` – the callback to delete.

**Raises** `ValueError` – if `callback` is not in the callback list of `DeviceDiscovered` event.

See also:

`XBeeNetwork.add_device_discovered_callback()`
`XBeeNetwork.add_discovery_process_finished_callback()`
`XBeeNetwork.del_discovery_process_finished_callback()`
**del_discovery_process_finished_callback** *(callback)*  
Deletes a callback for the callback list of **DiscoveryProcessFinished** event.

**Parameters**  
*callback* *(Function)* – the callback to delete.

**Raises**  
**ValueError** – if callback is not in the callback list of **DiscoveryProcessFinished** event.

**See also:**  
XBeeNetwork.add_discovery_process_finished_callback()  
XBeeNetwork.add_device_discovered_callback()  
XBeeNetwork.del_device_discovered_callback()

**del_end_discovery_scan_callback** *(callback)*  
Deletes a callback for the callback list of **EndDiscoveryScan**.

**Raises**  
**ValueError** – if callback is not in the callback list of **EndDiscoveryScan** event.

**See also:**  
XBeeNetwork.add_end_discovery_scan_callback()

**del_init_discovery_scan_callback** *(callback)*  
Deletes a callback for the callback list of **InitDiscoveryScan**.

**Raises**  
**ValueError** – if callback is not in the callback list of **InitDiscoveryScan** event.

**See also:**  
XBeeNetwork.add_init_discovery_scan_callback()

**del_network_modified_callback** *(callback)*  
Deletes a callback for the callback list of **digi.xbee.reader.NetworkModified**.

**Parameters**  
*callback* *(Function)* – the callback to delete.

**See also:**  
XBeeNetwork.add_network_modified_callback()

**discover_device** *(node_id)*  
Blocking method. Discovers and reports the first remote XBee device that matches the supplied identifier.

**Parameters**  
*node_id* *(String)* – the node identifier of the device to be discovered.

**Returns**  
the discovered remote XBee device with the given identifier, **None** if the timeout expires and the device was not found.
Return type: \texttt{RemoteXBeeDevice}

\texttt{discover\_devices}(\texttt{device\_id\_list})

Blocking method. Attempts to discover a list of devices and add them to the current network. This method does not guarantee that all devices of \texttt{device\_id\_list} will be found, even if they exist physically. This will depend on the node discovery operation (ND) and timeout.

\textbf{Parameters} \texttt{device\_id\_list} (List) – list of device IDs to discover.

\textbf{Returns} a list with the discovered devices. It may not contain all devices specified in \texttt{device\_id\_list}

Return type: List

\texttt{get\_connections}()

Returns a copy of the XBee connections.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

\textbf{Returns} A copy of the list of \texttt{Connection} for the network.

Return type: List

\texttt{get\_deep\_discovery\_options}()

Returns the deep discovery process options.

\textbf{Returns}

\((\texttt{digi.xbee.models.mode.NeighborDiscoveryMode}, \texttt{Boolean})\): Tuple containing:

\begin{itemize}
  \item \textbf{mode} (\texttt{digi.xbee.models.mode.NeighborDiscoveryMode}): Neighbor discovery mode, the way to perform the network discovery process.
  \item \textbf{remove\_nodes} (Boolean): True to remove nodes from the network if they were not discovered in the last scan, False otherwise.
\end{itemize}

Return type: Tuple

\textbf{See also:}

\begin{itemize}
  \item \texttt{digi.xbee.models.mode.NeighborDiscoveryMode}
  \item \texttt{XBeeNetwork.set\_deep\_discovery\_timeouts()}
  \item \texttt{XBeeNetwork.start\_discovery\_process()}
\end{itemize}

\texttt{get\_deep\_discovery\_timeouts}()

Gets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see \texttt{start\_discovery\_process()})

\textbf{Returns}

Tuple containing:

\begin{itemize}
  \item \textbf{node\_timeout} (Float): Maximum duration in seconds of the discovery process per node. This used to find a node neighbors. This timeout is highly dependent on the nature of the network:
    \begin{itemize}
      \item It should be greater than the highest NT (Node Discovery Timeout) of your network
    \end{itemize}
\end{itemize}
– and include enough time to let the message propagate depending on the
sleep cycle of your devices.

- **time_bw_nodes (Float)**: Time to wait between node neighbors requests.
  Use this setting not to saturate your network:
  – For ‘Cascade’ the number of seconds to wait after completion of the
    neighbor discovery process of the previous node.
  – For ‘Flood’ the minimum time to wait between each node’s neighbor re-
    quests.

- **time_bw_scans (Float)**: Time to wait before starting a new network scan.

**Return type** Tuple (Float, Float, Float)

**See also:**

XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

**get_device_by_16** *(x16bit_addr)*

Returns the XBee in the network whose 16-bit address matches the given one.

**Parameters** `x16bit_addr` (XBee16BitAddress) – The 16-bit address of the device
to be retrieved.

**Returns** the XBee device in the network or *None* if it is not found.

**Return type** AbstractXBeeDevice

**Raises** ValueError – if *x16bit_addr* is *None* or unknown.

**get_device_by_64** *(x64bit_addr)*

Returns the XBee in the network whose 64-bit address matches the given one.

**Parameters** `x64bit_addr` (XBee64BitAddress) – The 64-bit address of the device
to be retrieved.

**Returns** the XBee device in the network or *None* if it is not found.

**Return type** AbstractXBeeDevice

**Raises** ValueError – if *x64bit_addr* is *None* or unknown.

**get_device_by_node_id** *(node_id)*

Returns the XBee in the network whose node identifier matches the given one.

**Parameters** `node_id` (String) – The node identifier of the device to be retrieved.

**Returns** the XBee device in the network or *None* if it is not found.

**Return type** AbstractXBeeDevice

**Raises** ValueError – if *node_id* is *None*.

**get_devices** *

Returns a copy of the XBee devices list of the network.

If another XBee device is added to the list before the execution of this method, this XBee device will not be added to the list returned by this method.

**Returns** a copy of the XBee devices list of the network.
Return type  List

get_discovery_callbacks()  
Returns the API callbacks that are used in the device discovery process.  
This callbacks notify the user callbacks for each XBee device discovered.

Returns

- callback for generic devices discovery process,  callback for discovery specific XBee device ops.

Return type  Tuple (Function, Function)

get_discovery_options()  
Returns the network discovery process options.

Returns  the parameter value.

Return type  bytearray

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

get_discovery_timeout()  
Returns the network discovery timeout.

Returns  the network discovery timeout.

Return type  Float

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

get_node_connections(node)  
Returns the network connections with one of their ends node.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

Returns  List of Connection with node end.

Return type  List

get_number_devices()  
Returns the number of devices in the network.
Returns the number of devices in the network.

Return type: Integer

has_devices()
Returns whether there is any device in the network or not.

Returns: True if there is at least one device in the network, False otherwise.

Return type: Boolean

is_discovery_running()
Returns whether the discovery process is running or not.

Returns: True if the discovery process is running, False otherwise.

Return type: Boolean

remove_device(remote_xbee_device)
Removes the provided remote XBee device from the network.

Parameters

remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to be removed from the list.

Raises: ValueError – if the provided RemoteXBeeDevice is not in the network.

scan_counter
Returns the scan counter.

Returns: The scan counter.

Return type: Integer

set_deep_discovery_options(deep_mode=NeighborDiscoveryMode.CASCADE, del_not_discovered_nodes_in_last_scan=False)
Configures the deep discovery options with the given values. These options are only applicable for “deep” discovery (see start_discovery_process())

Parameters

- deep_mode (NeighborDiscoveryMode, optional, default=’Cascade’) – Neighbor discovery mode, the way to perform the network discovery process.
- del_not_discovered_nodes_in_last_scan (Boolean, optional, default=’False’) – True to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_deep_discovery_timeouts(node_timeout=None, time_bw_requests=None, time_bw_scans=None)
Sets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see start_discovery_process())

node_timeout (Float, optional, default=’None‘): Maximum duration in seconds of the discovery process used to find neighbors of a node. If None already configured timeouts are used.
time_bw_requests (Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS'): Time to wait between node neighbors requests. It must be between MIN_TIME_BETWEEN_REQUESTS and MAX_TIME_BETWEEN_REQUESTS seconds inclusive. Use this setting not to saturate your network:

- For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
- For ‘Flood’ the minimum time to wait between each node’s neighbor requests.

time_bw_scans (Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS'): Time to wait before starting a new network scan. It must be between MIN_TIME_BETWEEN_SCANS and MAX_TIME_BETWEEN_SCANS seconds inclusive.

Raises ValueError – if node_timeout, time_bw_requests or time_bw_scans are not between their corresponding limits.

See also:

XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_discovery_options (options)
Configures the discovery options (NO parameter) with the given value.

Parameters options (Set of DiscoveryOptions) – new discovery options, empty set to clear the options.

Raises

- ValueError – if options is None.
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
- InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

DiscoveryOptions

set_discovery_timeout (discovery_timeout)
Sets the discovery network timeout.

Parameters discovery_timeout (Float) – timeout in seconds.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBee device’s serial port is closed.
• *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• *ATCommandException* – if the response is not as expected.

• *ValueError* – if `discovery_timeout` is not between the allowed minimum and maximum values.

`start_discovery_process(\text{deep}=\text{False}, \text{n\_deep\_scans}=1)`

Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

• Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in `n\_deep\_scans`. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until `n\_deep\_scans` is satisfied.

  See `set\_deep\_discovery\_options()` to establish the way the network discovery process is performed.

• No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the ‘end’ packet is read.

  It may be that, after the timeout expires, there are devices that continue sending discovery packets to this XBee device. In this case, these devices will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

**Parameters**

• **deep** *(Boolean, optional, default=’False’)* – True for a deep network scan, looking for neighbors and their connections, False otherwise.

• **n\_deep\_scans** *(Integer, optional, default=1)* – Number of scans to perform before automatically stopping the discovery process. `SCAN\_TIL\_CANCEL` means the process will not be automatically stopped. Only applicable if `deep=True`.

**See also:**

`XBeeNetwork.add_device_discovered_callback()`  
`XBeeNetwork.add_discovery_process_finished_callback()`  
`XBeeNetwork.del_device_discovered_callback()`  
`XBeeNetwork.del_discovery_process_finished_callback()`  
`XBeeNetwork.get_deep_discovery_options()`  
`XBeeNetwork.set_deep_discovery_options()`

`stop_discovery_process()`

Stops the discovery process if it is running.

Note that DigiMesh/DigiPoint devices are blocked until the discovery time configured (NT parameter) has elapsed, so if you try to get/set any parameter during the discovery process you will receive a timeout exception.
class digi.xbee.devices.Raw802Network(device)
   Bases: digi.xbee.devices.XBeeNetwork

This class represents an 802.15.4 network.
The network allows the discovery of remote devices in the same network as the local one and stores them.

Class constructor. Instantiates a new Raw802Network.

   Parameters device (Raw802Device) – the local 802.15.4 device to get the network from.

   Raises ValueError – if device is None.

add_device_discovered_callback (callback)
   Adds a callback for the event DeviceDiscovered.

   Parameters callback (Function) – the callback. Receives one argument.
      • The discovered remote XBee device as a RemoteXBeeDevice

See also:
  XBeeNetwork.del_device_discovered_callback()
  XBeeNetwork.add_discovery_process_finished_callback()
  XBeeNetwork.del_discovery_process_finished_callback()

add_discovery_process_finished_callback (callback)
   Adds a callback for the event DiscoveryProcessFinished.

   Parameters callback (Function) – the callback. Receives one argument.
      • The event code as an Integer

See also:
  XBeeNetwork.del_discovery_process_finished_callback()
  XBeeNetwork.add_device_discovered_callback()
  XBeeNetwork.del_device_discovered_callback()

add_end_discovery_scan_callback (callback)
   Adds a callback for the event EndDiscoveryScan.

   Parameters callback (Function) – the callback. Receives two arguments.
      • Number of scan that has finished (starting with 1).
      • Total number of scans.

See also:
  XBeeNetwork.del_end_discovery_scan_callback()

add_if_not_exist (x64bit_addr=None, x16bit_addr=None, node_id=None)
   Adds an XBee device with the provided parameters if it does not exist in the current network.
   If the XBee device already exists, its data will be updated with the provided parameters that are not None.
Parameters

- `x64bit_addr` ([XBee64BitAddress](#), optional) – XBee device’s 64bit address. Optional.
- `x16bit_addr` ([XBee16BitAddress](#), optional) – XBee device’s 16bit address. Optional.
- `node_id` ([String](#), optional) – the node identifier of the XBee device. Optional.

Returns

the remote XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns the given XBee device without changes.

Return type `AbstractXBeeDevice`

**add_init_discovery_scan_callback** *(callback)*

Adds a callback for the event `InitDiscoveryScan`.

**Parameters**

- `callback` ([Function](#)) – the callback. Receives two arguments.
  - Number of scan to start (starting with 1).
  - Total number of scans.

See also:

`XBeeNetwork.del_init_discovery_scan_callback()`

**add_network_modified_callback** *(callback)*

Adds a callback for the event `digi.xbee.reader.NetworkModified`.

**Parameters**

- `callback` ([Function](#)) – the callback. Receives one argument.
  - The event type as a `NetworkEventType`
  - The reason of the event as a `NetworkEventReason`
  - The node added, updated or removed from the network as a `XBeeDevice` or `RemoteXBeeDevice`.

See also:

`XBeeNetwork.del_network_modified_callback()`

**add_remote** *(remote_xbee_device)*

Adds the provided remote XBee device to the network if it is not contained yet.

If the XBee device is already contained in the network, its data will be updated with the parameters of the XBee device that are not `None`.

**Parameters**

- `remote_xbee_device` ([RemoteXBeeDevice](#)) – the remote XBee device to add to the network.

Returns

the provided XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns it without changes.
Return type: `RemoteXBeeDevice`

**add_remotes** *(remote_xbee_devices)*

Adds a list of remote XBee devices to the network.

If any XBee device of the list is already contained in the network, its data will be updated with the parameters of the XBee device that are not `None`.

**Parameters**

- `remote_xbee_devices` *(List)* – the list of `RemoteXBeeDevice` to add to the network.

**clear** *

Removes all the remote XBee devices from the network.

**del_device_discovered_callback** *(callback)*

Deletes a callback for the callback list of `DeviceDiscovered` event.

**Parameters**

- `callback` *(Function)* – the callback to delete.

**Raises** `ValueError` – if `callback` is not in the callback list of `DeviceDiscovered` event.

See also:

- `XBeeNetwork.add_device_discovered_callback()`
- `XBeeNetwork.add_discovery_process_finished_callback()`
- `XBeeNetwork.del_discovery_process_finished_callback()`

**del_discovery_process_finished_callback** *(callback)*

Deletes a callback for the callback list of `DiscoveryProcessFinished` event.

**Parameters**

- `callback` *(Function)* – the callback to delete.

**Raises** `ValueError` – if `callback` is not in the callback list of `DiscoveryProcessFinished` event.

See also:

- `XBeeNetwork.add_discovery_process_finished_callback()`
- `XBeeNetwork.add_device_discovered_callback()`
- `XBeeNetwork.del_device_discovered_callback()`

**del_end_discovery_scan_callback** *(callback)*

Deletes a callback for the callback list of `EndDiscoveryScan`.

**Raises** `ValueError` – if `callback` is not in the callback list of `EndDiscoveryScan` event.

See also:

- `XBeeNetwork.add_end_discovery_scan_callback()`

**del_init_discovery_scan_callback** *(callback)*

Deletes a callback for the callback list of `InitDiscoveryScan`.
Raises `ValueError` – if `callback` is not in the callback list of `InitDiscoveryScan` event.

See also:

```
XBeeNetwork.add_init_discovery_scan_callback()
```

del_network_modified_callback (`callback`) Deletes a callback for the callback list of `digi.xbee.reader.NetworkModified`.

Parameters `callback` (Function) – the callback to delete.

See also:

```
XBeeNetwork.add_network_modified_callback()
```

discover_device (`node_id`) Blocking method. Discovers and reports the first remote XBee device that matches the supplied identifier.

Parameters `node_id` (String) – the node identifier of the device to be discovered.

Returns

the discovered remote XBee device with the given identifier, None if the timeout expires and the device was not found.

Return type `RemoteXBeeDevice`

discover_devices (`device_id_list`) Blocking method. Attempts to discover a list of devices and add them to the current network.

This method does not guarantee that all devices of `device_id_list` will be found, even if they exist physically. This will depend on the node discovery operation (ND) and timeout.

Parameters `device_id_list` (List) – list of device IDs to discover.

Returns a list with the discovered devices. It may not contain all devices specified in `device_id_list`

Return type `List`

get_connections () Returns a copy of the XBee connections.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

Returns A copy of the list of `Connection` for the network.

Return type `List`

get_deep_discovery_options () Returns the deep discovery process options.

Returns

```
(digi.xbee.models.mode.NeighborDiscoveryMode, Boolean): Tuple containing:
```

mode (\texttt{digi.xbee.models.mode.NeighborDiscoveryMode}): Neighbor discovery mode, the way to perform the network discovery process.

remove_nodes (Boolean): \texttt{True} to remove nodes from the network if they were not discovered in the last scan, \texttt{False} otherwise.

Return type Tuple

See also:

\texttt{digi.xbee.models.mode.NeighborDiscoveryMode}
\texttt{XBeeNetwork.set_deep_discovery_timeouts()}
\texttt{XBeeNetwork.start_discovery_process()}

\texttt{get\_deep\_discovery\_timeouts()} 
Gets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see \texttt{start\_discovery\_process()})

Returns Tuple containing:

- node_timeout (Float): Maximum duration in seconds of the discovery process per node. This used to find a node neighbors. This timeout is highly dependent on the nature of the network:
  - It should be greater than the highest NT (Node Discovery Timeout) of your network
  - and include enough time to let the message propagate depending on the sleep cycle of your devices.

- time\_bw\_nodes (Float): Time to wait between node neighbors requests. Use this setting not to saturate your network:
  - For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For ‘Flood’ the minimum time to wait between each node’s neighbor requests.

- time\_bw\_scans (Float): Time to wait before starting a new network scan.

Return type Tuple (Float, Float, Float)

See also:

\texttt{XBeeNetwork.set_deep_discovery_timeouts()}
\texttt{XBeeNetwork.start_discovery_process()}

\texttt{get\_device\_by\_16(x16bit\_addr)} 
Returns the XBee in the network whose 16-bit address matches the given one.

Parameters x16bit\_addr (XBee16BitAddress) – The 16-bit address of the device to be retrieved.

Returns the XBee device in the network or \texttt{None} if it is not found.

Return type AbstractXBeeDevice
```python
Raises ValueError – if x16bit_addr is None or unknown.

def get_device_by_64(x64bit_addr)
    Returns the XBee in the network whose 64-bit address matches the given one.
    Parameters x64bit_addr (XBee64BitAddress) – The 64-bit address of the device to be retrieved.
    Returns the XBee device in the network or None if it is not found.
    Return type AbstractXBeeDevice
    Raises ValueError – if x64bit_addr is None or unknown.

def get_device_by_node_id(node_id)
    Returns the XBee in the network whose node identifier matches the given one.
    Parameters node_id (String) – The node identifier of the device to be retrieved.
    Returns the XBee device in the network or None if it is not found.
    Return type AbstractXBeeDevice
    Raises ValueError – if node_id is None.

def get_devices()
    Returns a copy of the XBee devices list of the network.
    If another XBee device is added to the list before the execution of this method, this XBee device will not be added to the list returned by this method.
    Returns a copy of the XBee devices list of the network.
    Return type List

def get_discovery_callbacks()
    Returns the API callbacks that are used in the device discovery process.
    This callbacks notify the user callbacks for each XBee device discovered.
    Returns
        callback for generic devices discovery process, callback for discovery specific XBee device ops.
    Return type Tuple (Function, Function)

def get_discovery_options()
    Returns the network discovery process options.
    Returns the parameter value.
    Return type Bytearray
    Raises
        • TimeoutException – if the response is not received before the read timeout expires.
        • XBeeException – if the XBee device’s serial port is closed.
        • InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
        • ATCommandException – if the response is not as expected.
```
get_discovery_timeout()  
Returns the network discovery timeout.

Returns  the network discovery timeout.

Return type  Float

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

get_node_connections(node)  
Returns the network connections with one of their ends node.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

Returns  List of `Connection` with `node` end.

Return type  List

get_number_devices()  
Returns the number of devices in the network.

Returns  the number of devices in the network.

Return type  Integer

has_devices()  
Returns whether there is any device in the network or not.

Returns  True if there is at least one device in the network, `False` otherwise.

Return type  Boolean

is_discovery_running()  
Returns whether the discovery process is running or not.

Returns  `True` if the discovery process is running, `False` otherwise.

Return type  Boolean

remove_device(remote_xbee_device)  
Removes the provided remote XBee device from the network.

Parameters  

remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to be removed from the list.

Raises `ValueError` – if the provided `RemoteXBeeDevice` is not in the network.

scan_counter  
Returns the scan counter.

Returns  The scan counter.

Return type  Integer
**set_deep_discovery_options**

```python
(set_deep_discovery_options(deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>, del_not_discovered_nodes_in_last_scan=False))
```

Configures the deep discovery options with the given values. These options are only applicable for “deep” discovery (see `start_discovery_process()`)

**Parameters**

- **deep_mode** *(NeighborDiscoveryMode, optional, default='NeighborDiscoveryMode.CASCADE')* - Neighbor discovery mode, the way to perform the network discovery process.

- **del_not_discovered_nodes_in_last_scan** *(Boolean, optional, default=``False``)* - True to remove nodes from the network if they were not discovered in the last scan.

See also:

- `digi.xbee.models.mode.NeighborDiscoveryMode`
- `XBeeNetwork.get_deep_discovery_timeouts()`
- `XBeeNetwork.start_discovery_process()`

**set_deep_discovery_timeouts**

```python
(set_deep_discovery_timeouts(node_timeout=None, time_bw_requests=None, time_bw_scans=None))
```

Sets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see `start_discovery_process()`)

- **node_timeout** *(Float, optional, default='None')*: Maximum duration in seconds of the discovery process used to find neighbors of a node. If `None` already configured timeouts are used.

- **time_bw_requests** *(Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS')*: Time to wait between node neighbors requests. It must be between `MIN_TIME_BETWEEN_REQUESTS` and `MAX_TIME_BETWEEN_REQUESTS` seconds inclusive. Use this setting not to saturate your network:
  - For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For ‘Flood’ the minimum time to wait between each node’s neighbor requests.

- **time_bw_scans** *(Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS')*: Time to wait before starting a new network scan. It must be between `MIN_TIME_BETWEEN_SCANS` and `MAX_TIME_BETWEEN_SCANS` seconds inclusive.

  Raises **ValueError** - if `node_timeout`, `time_bw_requests` or `time_bw_scans` are not between their corresponding limits.

See also:

- `XBeeNetwork.get_deep_discovery_timeouts()`
- `XBeeNetwork.start_discovery_process()`

**set_discovery_options** *(options)*

Configures the discovery options (NO parameter) with the given value.
**Parameters** options (Set of DiscoveryOptions) – new discovery options, empty set to clear the options.

**Raises**

- **ValueError** – if options is None.
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

DiscoveryOptions

**set_discovery_timeout** (discovery_timeout)

Sets the discovery network timeout.

**Parameters** discovery_timeout (Float) – timeout in seconds.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **ValueError** – if discovery_timeout is not between the allowed minimum and maximum values.

**start_discovery_process** (deep=False, n_deep_scans=1)

Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

- Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in n_deep_scans. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until n_deep_scans is satisfied.

  See **set_deep_discovery_options()** to establish the way the network discovery process is performed.

- No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the ‘end’ packet is read.
It may be that, after the timeout expires, there are devices that continue sending discovery packets to this XBee device. In this case, these devices will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

**Parameters**

- **deep** *(Boolean, optional, default=`False`) – True for a deep network scan, looking for neighbors and their connections, False otherwise.*

- **n_deep_scans** *(Integer, optional, default=1) – Number of scans to perform before automatically stopping the discovery process. SCAN_TIL_CANCEL means the process will not be automatically stopped. Only applicable if deep=True.*

**See also:**

- XBeeNetwork.add_device_discovered_callback()
- XBeeNetwork.add_discovery_process_finished_callback()
- XBeeNetwork.del_device_discovered_callback()
- XBeeNetwork.del_discovery_process_finished_callback()
- XBeeNetwork.get_deep_discovery_options()
- XBeeNetwork.set_deep_discovery_options()

**stop_discovery_process()**

Stops the discovery process if it is running.

Note that DigiMesh/DigiPoint devices are blocked until the discovery time configured (NT parameter) has elapsed, so if you try to get/set any parameter during the discovery process you will receive a timeout exception.

```python
class digi.xbee.devices.DigiMeshNetwork(device)
Bases: digi.xbee.devices.XBeeNetwork
```

This class represents a DigiMesh network.

The network allows the discovery of remote devices in the same network as the local one and stores them.

Class constructor. Instantiates a new DigiMeshNetwork.

**Parameters** **device** *(DigiMeshDevice) – the local DigiMesh device to get the network from.*

**Raises** **ValueError** – if device is None.

**add_device_discovered_callback** *(callback)*

Adds a callback for the event DeviceDiscovered.

**Parameters** **callback** *(Function) – the callback. Receives one argument.*

- The discovered remote XBee device as a RemoteXBeeDevice
add_discovery_process_finished_callback \(\text{callback}\)

Adds a callback for the event \text{DiscoveryProcessFinished}.

**Parameters**

\text{callback} \(\text{Function}\) – the callback. Receives one argument.

- The event code as an \text{Integer}

See also:

\begin{verbatim}
XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()
\end{verbatim}

add_end_discovery_scan_callback \(\text{callback}\)

Adds a callback for the event \text{EndDiscoveryScan}.

**Parameters**

\text{callback} \(\text{Function}\) – the callback. Receives two arguments.

- Number of scan that has finished (starting with 1).
- Total number of scans.

See also:

\begin{verbatim}
XBeeNetwork.del_end_discovery_scan_callback()
\end{verbatim}

add_if_not_exist \(x64bit_addr=\text{None}, x16bit_addr=\text{None}, node_id=\text{None}\)

Adds an XBee device with the provided parameters if it does not exist in the current network. If the XBee device already exists, its data will be updated with the provided parameters that are not \text{None}.

**Parameters**

- \text{x64bit_addr} (XBee64BitAddress, optional) – XBee device’s 64bit address. Optional.
- \text{x16bit_addr} (XBee16BitAddress, optional) – XBee device’s 16bit address. Optional.
- \text{node_id} (String, optional) – the node identifier of the XBee device. Optional.

**Returns**

the remote XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns the given XBee device without changes.

**Return type** \text{AbstractXBeeDevice}

add_init_discovery_scan_callback \(\text{callback}\)

Adds a callback for the event \text{InitDiscoveryScan}.

**Parameters**

\text{callback} \(\text{Function}\) – the callback. Receives two arguments.

- Number of scan to start (starting with 1).
- Total number of scans.

See also:
XBeeNetwork.del_init_discovery_scan_callback()

add_network_modified_callback(callback)
Adds a callback for the event \texttt{digi.xbee.reader.NetworkModified}.

\textbf{Parameters} \texttt{callback} (Function) – the callback. Receives one argument.

- The event type as a \texttt{NetworkEventType}
- The reason of the event as a \texttt{NetworkEventReason}
- The node added, updated or removed from the network as a \texttt{XBeeDevice} or \texttt{RemoteXBeeDevice}.

See also:

XBeeNetwork.del_network_modified_callback()

add_remotes(remote_xbee_devices)
Adds a list of remote XBee devices to the network.

If any XBee device of the list is already contained in the network, its data will be updated with the parameters of the XBee device that are not None.

\textbf{Parameters} \texttt{remote_xbee_devices} (List) – the list of \texttt{RemoteXBeeDevice} to add to the network.

\textbf{Return type} \texttt{RemoteXBeeDevice}

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_discovery_process_finished_callback()
**del_discovery_process_finished_callback** *(callback)*

Deletes a callback for the callback list of *DiscoveryProcessFinished* event.

**Parameters**

- **callback** *(Function)* – the callback to delete.

**Raises** *

- **ValueError** – if *callback* is not in the callback list of *

  *DiscoveryProcessFinished* event.

**See also:**

- XBeeNetwork.add_discovery_process_finished_callback()
- XBeeNetwork.add_device_discovered_callback()
- XBeeNetwork.del_device_discovered_callback()

**del_end_discovery_scan_callback** *(callback)*

Deletes a callback for the callback list of *EndDiscoveryScan*.

**Raises** *

- **ValueError** – if *callback* is not in the callback list of *

  *EndDiscoveryScan* event.

**See also:**

- XBeeNetwork.add_end_discovery_scan_callback()

**del_init_discovery_scan_callback** *(callback)*

Deletes a callback for the callback list of *InitDiscoveryScan*.

**Raises** *

- **ValueError** – if *callback* is not in the callback list of *

  *InitDiscoveryScan* event.

**See also:**

- XBeeNetwork.add_init_discovery_scan_callback()

**del_network_modified_callback** *(callback)*

Deletes a callback for the callback list of *digi.xbee.reader.NetworkModified*.

**Parameters**

- **callback** *(Function)* – the callback to delete.

**See also:**

- XBeeNetwork.add_network_modified_callback()

**discover_device** *(node_id)*

Blocking method. Discovers and reports the first remote XBee device that matches the supplied identifier.

**Parameters**

- **node_id** *(String)* – the node identifier of the device to be discovered.

**Returns**

- the discovered remote XBee device with the given identifier, *None* if the timeout expires and the device was not found.
discover_devices(device_id_list)

Blocking method. Attempts to discover a list of devices and add them to the current network.

This method does not guarantee that all devices of device_id_list will be found, even if they exist physically. This will depend on the node discovery operation (ND) and timeout.

Parameters device_id_list (List) – list of device IDs to discover.

Returns a list with the discovered devices. It may not contain all devices specified in device_id_list

Return type List

get_connections()

Returns a copy of the XBee connections.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

Returns A copy of the list of Connection for the network.

Return type List

get_deep_discovery_options()

Returns the deep discovery process options.

Returns 
(digi.xbee.models.mode.NeighborDiscoveryMode, Boolean): Tuple containing:

• mode (digi.xbee.models.mode.NeighborDiscoveryMode): Neighbor discovery mode, the way to perform the network discovery process.

• remove_nodes (Boolean): True to remove nodes from the network if they were not discovered in the last scan, False otherwise.

Return type Tuple

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_deep_discovery_timeouts()

Gets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see start_discovery_process())

Returns 
 Tuple containing:

• node_timeout (Float): Maximum duration in seconds of the discovery process per node. This used to find a node neighbors. This timeout is highly dependent on the nature of the network:

  – It should be greater than the highest NT (Node Discovery Timeout) of your network
– and include enough time to let the message propagate depending on the
sleep cycle of your devices.

- **time_bw_nodes (Float): Time to wait between node neighbors requests.**
  Use this setting not to saturate your network:
  – For ‘Cascade’ the number of seconds to wait after completion of the
    neighbor discovery process of the previous node.
  – For ‘Flood’ the minimum time to wait between each node’s neighbor re-
    quests.

- **time_bw_scans (Float): Time to wait before starting a new network scan.**

  **Return type** Tuple (Float, Float, Float)

  **See also:**

  - XBeeNetwork.set_deep_discovery_timeouts()
  - XBeeNetwork.start_discovery_process()

  **get_device_by_16 (x16bit_addr)**
  Returns the XBee in the network whose 16-bit address matches the given one.

  **Parameters** x16bit_addr (XBee16BitAddress) – The 16-bit address of the device
to be retrieved.

  **Returns** the XBee device in the network or None if it is not found.

  **Return type** AbstractXBeeDevice

  **Raises** ValueError – if x16bit_addr is None or unknown.

  **get_device_by_64 (x64bit_addr)**
  Returns the XBee in the network whose 64-bit address matches the given one.

  **Parameters** x64bit_addr (XBee64BitAddress) – The 64-bit address of the device
to be retrieved.

  **Returns** the XBee device in the network or None if it is not found.

  **Return type** AbstractXBeeDevice

  **Raises** ValueError – if x64bit_addr is None or unknown.

  **get_device_by_node_id (node_id)**
  Returns the XBee in the network whose node identifier matches the given one.

  **Parameters** node_id (String) – The node identifier of the device to be retrieved.

  **Returns** the XBee device in the network or None if it is not found.

  **Return type** AbstractXBeeDevice

  **Raises** ValueError – if node_id is None.

  **get_devices ()**
  Returns a copy of the XBee devices list of the network.

  If another XBee device is added to the list before the execution of this method, this XBee device will not
be added to the list returned by this method.

  **Returns** a copy of the XBee devices list of the network.
Return type  List

get_discovery_callbacks()  
Returns the API callbacks that are used in the device discovery process. This callbacks notify the user callbacks for each XBee device discovered.

Returns

callback for generic devices discovery process,  callback for discovery specific XBee device ops.

Return type  Tuple (Function, Function)

get_discovery_options()  
Returns the network discovery process options.

Returns  the parameter value.

Return type  bytearray

Raises

•  TimeoutException – if the response is not received before the read timeout expires.
•  XBeeException – if the XBee device’s serial port is closed.
•  InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
•  ATCommandException – if the response is not as expected.

get_discovery_timeout()  
Returns the network discovery timeout.

Returns  the network discovery timeout.

Return type  Float

Raises

•  TimeoutException – if the response is not received before the read timeout expires.
•  XBeeException – if the XBee device’s serial port is closed.
•  InvalidOperatingModeException – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
•  ATCommandException – if the response is not as expected.

get_node_connections(node)  
Returns the network connections with one of their ends node.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

Returns  List of Connection with node end.

Return type  List

get_number_devices()  
Returns the number of devices in the network.
Returns the number of devices in the network.

Return type  Integer

has_devices()
Returns whether there is any device in the network or not.

Returns  True if there is at least one device in the network, False otherwise.

Return type  Boolean

is_discovery_running()
Returns whether the discovery process is running or not.

Returns  True if the discovery process is running, False otherwise.

Return type  Boolean

remove_device(remote_xbee_device)
Removes the provided remote XBee device from the network.

Parameters  remote_xbee_device (RemoteXBeeDevice) – the remote XBee device to be removed from the list.

Raises  ValueError – if the provided RemoteXBeeDevice is not in the network.

scan_counter
Returns the scan counter.

Returns  The scan counter.

Return type  Integer

set_deep_discovery_options(deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>, del_not_discovered_nodes_in_last_scan=False)
Configures the deep discovery options with the given values. These options are only applicable for “deep” discovery (see start_discovery_process())

Parameters

• deep_mode  (NeighborDiscoveryMode, optional, default=’NeighborDiscoveryMode.CASCADE’) – Neighbor discovery mode, the way to perform the network discovery process.

• del_not_discovered_nodes_in_last_scan  (Boolean, optional, default=‘False’) – True to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_deep_discovery_timeouts(node_timeout=None, time_bw_requests=None, time_bw_scans=None)
Sets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see start_discovery_process())

node_timeout (Float, optional, default=’None‘): Maximum duration in seconds of the discovery process used to find neighbors of a node. If None already configured timeouts are used.
time_bw_requests (Float, optional, default=‘DEFAULT_TIME_BETWEEN_REQUESTS‘): Time to wait between node neighbors requests. It must be between MIN_TIME_BETWEEN_REQUESTS and MAX_TIME_BETWEEN_REQUESTS seconds inclusive. Use this setting not to saturate your network:

- For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
- For ‘Flood’ the minimum time to wait between each node’s neighbor requests.

time_bw_scans (Float, optional, default=‘DEFAULT_TIME_BETWEEN_SCANS‘): Time to wait before starting a new network scan. It must be between MIN_TIME_BETWEEN_SCANS and MAX_TIME_BETWEEN_SCANS seconds inclusive.

Raises ValueError – if node_timeout, time_bw_requests or time_bw_scans are not between their corresponding limits.

See also:

XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_discovery_options (options)
Configures the discovery options (NO parameter) with the given value.

Parameters options (Set of DiscoveryOptions) – new discovery options, empty set to clear the options.

Raises

- ValueError – if options is None.
- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBees device’s serial port is closed.
- InvalidOperatingModeException – if the XBees device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – if the response is not as expected.

See also:

DiscoveryOptions

set_discovery_timeout (discovery_timeout)
Sets the discovery network timeout.

Parameters discovery_timeout (Float) – timeout in seconds.

Raises

- TimeoutException – if the response is not received before the read timeout expires.
- XBeeException – if the XBees device’s serial port is closed.
• **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

• **ATCommandException** – if the response is not as expected.

• **ValueError** – if `discovery_timeout` is not between the allowed minimum and maximum values.

`start_discovery_process((deep=False, n_deep_scans=1))`

Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

• Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in `n_deep_scans`. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until `n_deep_scans` is satisfied.

  See `set_deep_discovery_options()` to establish the way the network discovery process is performed.

• No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the ‘end’ packet is read.

  It may be that, after the timeout expires, there are devices that continue sending discovery packets to this XBee device. In this case, these devices will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

**Parameters**

• **deep** *(Boolean, optional, default=’False’)* – True for a deep network scan, looking for neighbors and their connections. False otherwise.

• **n_deep_scans** *(Integer, optional, default=1)* – Number of scans to perform before automatically stopping the discovery process. SCAN_TIL_CANCEL means the process will not be automatically stopped. Only applicable if deep=True.

See also:

`XBeeNetwork.add_device_discovered_callback()`
`XBeeNetwork.add_discovery_process_finished_callback()`
`XBeeNetwork.del_device_discovered_callback()`
`XBeeNetwork.del_discovery_process_finished_callback()`
`XBeeNetwork.get_deep_discovery_options()`
`XBeeNetwork.set_deep_discovery_options()`

`stop_discovery_process()`

Stops the discovery process if it is running.

Note that DigiMesh/DigiPoint devices are blocked until the discovery time configured (NT parameter) has elapsed, so if you try to get/set any parameter during the discovery process you will receive a timeout exception.
class digi.xbee.devices.DigiPointNetwork(device)
Bases: digi.xbee.devices.XBeeNetwork

This class represents a DigiPoint network.

The network allows the discovery of remote devices in the same network as the local one and stores them.

Class constructor. Instantiates a new DigiPointNetwork.

Parameters device (DigiPointDevice) – the local DigiPoint device to get the network from.

Raises ValueError – if device is None.

add_device_discovered_callback (callback)
Adds a callback for the event DeviceDiscovered.

Parameters callback (Function) – the callback. Receives one argument.

• The discovered remote XBee device as a RemoteXBeeDevice

See also:

XBeeNetwork.del_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback() 
XBeeNetwork.del_discovery_process_finished_callback()

add_discovery_process_finished_callback (callback)
Adds a callback for the event DiscoveryProcessFinished.

Parameters callback (Function) – the callback. Receives one argument.

• The event code as an Integer

See also:

XBeeNetwork.del_discovery_process_finished_callback() 
XBeeNetwork.add_device_discovered_callback() 
XBeeNetwork.del_discovery_process_finished_callback()

add_end_discovery_scan_callback (callback)
Adds a callback for the event EndDiscoveryScan.

Parameters callback (Function) – the callback. Receives two arguments.

• Number of scan that has finished (starting with 1).

• Total number of scans.

See also:

XBeeNetwork.del_end_discovery_scan_callback()

add_if_not_exist (x64bit_addr=None, x16bit_addr=None, node_id=None)
Adds an XBee device with the provided parameters if it does not exist in the current network.

If the XBee device already exists, its data will be updated with the provided parameters that are not None.
Parameters

- `x64bit_addr` (XBee64BitAddress, optional) – XBee device’s 64bit address. Optional.
- `x16bit_addr` (XBee16BitAddress, optional) – XBee device’s 16bit address. Optional.
- `node_id` (String, optional) – the node identifier of the XBee device. Optional.

Returns

the remote XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns the given XBee device without changes.

Return type `AbstractXBeeDevice`

`add_init_discovery_scan_callback` (*callback*)

Adds a callback for the event `InitDiscoveryScan`.

Parameters `callback` (Function) – the callback. Receives two arguments.

- Number of scan to start (starting with 1).
- Total number of scans.

See also:

`XBeeNetwork.del_init_discovery_scan_callback()`

`add_network_modified_callback` (*callback*)

Adds a callback for the event `digi.xbee.reader.NetworkModified`.

Parameters `callback` (Function) – the callback. Receives one argument.

- The event type as a `NetworkEventType`
- The reason of the event as a `NetworkEventReason`
- The node added, updated or removed from the network as a `XBeeDevice` or `RemoteXBeeDevice`.

See also:

`XBeeNetwork.del_network_modified_callback()`

`add_remote` (*remote_xbee_device*)

Adds the provided remote XBee device to the network if it is not contained yet.

If the XBee device is already contained in the network, its data will be updated with the parameters of the XBee device that are not `None`.

Parameters `remote_xbee_device` (RemoteXBeeDevice) – the remote XBee device to add to the network.

Returns

the provided XBee device with the updated parameters. If the XBee device was not in the list yet, this method returns it without changes.
Return type RemoteXBeeDevice

add_remotes(remote_xbee_devices)

Adds a list of remote XBees to the network.

If any XBees of the list is already contained in the network, its data will be updated with the parameters of the XBees that are not None.

Parameters remote_xbee_devices (List) – the list of RemoteXBeeDevice to add to the network.

clear()

Removes all the remote XBees from the network.

del_device_discovered_callback(callback)

Deletes a callback for the callback list of DeviceDiscovered event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of DeviceDiscovered event.

See also:

XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_discovery_process_finished_callback()

del_discovery_process_finished_callback(callback)

Deletes a callback for the callback list of DiscoveryProcessFinished event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of DiscoveryProcessFinished event.

See also:

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

del_end_discovery_scan_callback(callback)

Deletes a callback for the callback list of EndDiscoveryScan.

Raises ValueError – if callback is not in the callback list of EndDiscoveryScan event.

See also:

XBeeNetwork.add_end_discovery_scan_callback()

del_init_discovery_scan_callback(callback)

Deletes a callback for the callback list of InitDiscoveryScan.
Raises ValueError – if callback is not in the callback list of InitDiscoveryScan event.

See also:

XBeeNetwork.add_init_discovery_scan_callback()

del_network_modified_callback (callback)
Deletes a callback for the callback list of digi.xbee.reader.NetworkModified.

Parameters callback (Function) – the callback to delete.

See also:

XBeeNetwork.add_network_modified_callback()

discover_device (node_id)
Blocking method. Discovers and reports the first remote XBee device that matches the supplied identifier.

Parameters node_id (String) – the node identifier of the device to be discovered.

Returns

the discovered remote XBee device with the given identifier, None if the timeout expires and the device was not found.

Return type RemoteXBeeDevice

discover_devices (device_id_list)
Blocking method. Attempts to discover a list of devices and add them to the current network.

This method does not guarantee that all devices of device_id_list will be found, even if they exist physically. This will depend on the node discovery operation (ND) and timeout.

Parameters device_id_list (List) – list of device IDs to discover.

Returns a list with the discovered devices. It may not contain all devices specified in device_id_list

Return type List

get_connections ()
Returns a copy of the XBee connections.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

Returns A copy of the list of Connection for the network.

Return type List

get_deep_discovery_options ()
Returns the deep discovery process options.

Returns

(digi.xbee.models.mode.NeighborDiscoveryMode, Boolean): Tuple containing:
• mode *(digixbeemodels.mode.NeighborDiscoveryMode)*: Neighbor discovery mode, the way to perform the network discovery process.

• remove_nodes (Boolean): True to remove nodes from the network if they were not discovered in the last scan, False otherwise.

Return type Tuple

See also:

digixbeemodels.mode.NeighborDiscoveryMode
XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_deep_discovery_timeouts()

Gets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see start_discovery_process())

Returns

Tuple containing:

• node_timeout (Float): Maximum duration in seconds of the discovery process per node. This used to find a node neighbors. This timeout is highly dependent on the nature of the network:
  – It should be greater than the highest NT (Node Discovery Timeout) of your network
  – and include enough time to let the message propagate depending on the sleep cycle of your devices.

• time_bw_nodes (Float): Time to wait between node neighbors requests. Use this setting not to saturate your network:
  – For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  – For ‘Flood’ the minimum time to wait between each node’s neighbor requests.

• time_bw_scans (Float): Time to wait before starting a new network scan.

Return type Tuple (Float, Float, Float)

See also:

XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_device_by_16(x16bit_addr)

Returns the XBee in the network whose 16-bit address matches the given one.

Parameters x16bit_addr (XBeel16BitAddress) – The 16-bit address of the device to be retrieved.

Returns the XBee device in the network or None if it is not found.

Return type AbstractXBeeDevice
XBee Python Library Documentation, Release 1.3.0

Raises `ValueError` – if `x16bit_addr` is `None` or unknown.

### get_device_by_64(x64bit_addr)
Returns the XBee in the network whose 64-bit address matches the given one.

**Parameters**
- `x64bit_addr` (XBee64BitAddress) – The 64-bit address of the device to be retrieved.

**Returns**
- the XBee device in the network or `None` if it is not found.

**Return type**
- `AbstractXBeeDevice`

### get_device_by_node_id(node_id)
Returns the XBee in the network whose node identifier matches the given one.

**Parameters**
- `node_id` (String) – The node identifier of the device to be retrieved.

**Returns**
- the XBee device in the network or `None` if it is not found.

**Return type**
- `AbstractXBeeDevice`

### get_devices()
Returns a copy of the XBee devices list of the network.

If another XBee device is added to the list before the execution of this method, this XBee device will not be added to the list returned by this method.

**Returns**
- a copy of the XBee devices list of the network.

**Return type**
- `List`

### get_discovery_callbacks()
Returns the API callbacks that are used in the device discovery process.

This callbacks notify the user callbacks for each XBee device discovered.

**Returns**
- `callback for generic devices discovery process`, `callback for discovery specific XBee device ops`.

**Return type**
- `Tuple (Function, Function)`

### get_discovery_options()
Returns the network discovery process options.

**Returns**
- the parameter value.

**Return type**
- `Bytearray`

### Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.
**get_discovery_timeout()**

Returns the network discovery timeout.

- **Returns** the network discovery timeout.
- **Return type** Float
- **Raises**
  - *TimeoutException* – if the response is not received before the read timeout expires.
  - *XBeeException* – if the XBee device’s serial port is closed.
  - *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
  - *ATCommandException* – if the response is not as expected.

**get_node_connections(node)**

Returns the network connections with one of their ends `node`.

If a new connection is added to the list after the execution of this method, this connection is not added to the list returned by this method.

- **Returns** List of `Connection` with `node` end.
- **Return type** List

**get_number_devices()**

Returns the number of devices in the network.

- **Returns** the number of devices in the network.
- **Return type** Integer

**has_devices()**

Returns whether there is any device in the network or not.

- **Returns** True if there is at least one device in the network, False otherwise.
- **Return type** Boolean

**is_discovery_running()**

Returns whether the discovery process is running or not.

- **Returns** True if the discovery process is running, False otherwise.
- **Return type** Boolean

**remove_device(remote_xbee_device)**

Removes the provided remote XBee device from the network.

- **Parameters** `remote_xbee_device` (`RemoteXBeeDevice`) – the remote XBee device to be removed from the list.
- **Raises** *ValueError* – if the provided `RemoteXBeeDevice` is not in the network.

**scan_counter**

Returns the scan counter.

- **Returns** The scan counter.
- **Return type** Integer
set_deep_discovery_options (deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>, del_not_discovered_nodes_in_last_scan=False)

Configures the deep discovery options with the given values. These options are only applicable for “deep” discovery (see start_discovery_process())

Parameters

- **deep_mode** (NeighborDiscoveryMode, optional, default=’NeighborDiscoveryMode.CASCADE’) – Neighbor discovery mode, the way to perform the network discovery process.
- **del_not_discovered_nodes_in_last_scan** (Boolean, optional, default=’False’) – True to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_deep_discovery_timeouts (node_timeout=None, time_bw_requests=None, time_bw_scans=None)

Sets deep discovery network timeouts. These timeouts are only applicable for “deep” discovery (see start_discovery_process())

- **node_timeout** (Float, optional, default=’None‘): Maximum duration in seconds of the discovery process used to find neighbors of a node. If None already configured timeouts are used.
- **time_bw_requests** (Float, optional, default=’DEFAULT_TIME_BETWEEN_REQUESTS‘): Time to wait between node neighbors requests. It must be between MIN_TIME_BETWEEN_REQUESTS and MAX_TIME_BETWEEN_REQUESTS seconds inclusive. Use this setting not to saturate your network:
  - For ‘Cascade’ the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For ‘Flood’ the minimum time to wait between each node’s neighbor requests.
- **time_bw_scans** (Float, optional, default=’DEFAULT_TIME_BETWEEN_SCANS‘): Time to wait before starting a new network scan. It must be between MIN_TIME_BETWEEN_SCANS and MAX_TIME_BETWEEN_SCANS seconds inclusive.

Raises ValueError – if node_timeout, time_bw_requests or time_bw_scans are not between their corresponding limits.

See also:

XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_discovery_options (options)

Configures the discovery options (NO parameter) with the given value.
**Parameters options** (Set of DiscoveryOptions) – new discovery options, empty set to clear the options.

**Raises**

- **ValueError** – if options is None.
- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.

See also:

**DiscoveryOptions**

**set_discovery_timeout** (**discovery_timeout**)  
Sets the discovery network timeout.

**Parameters**  
**discovery_timeout** (**Float**) – timeout in seconds.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- **ATCommandException** – if the response is not as expected.
- **ValueError** – if discovery_timeout is not between the allowed minimum and maximum values.

**start_discovery_process** (**deep=False, n_deep_scans=1**)  
Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

- Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in n_deep_scans. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until n_deep_scans is satisfied.

  See **set_deep_discovery_options()** to establish the way the network discovery process is performed.

- No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the ‘end’ packet is read.
It may be that, after the timeout expires, there are devices that continue sending discovery packets to this XBee device. In this case, these devices will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

**Parameters**

- **deep** *(Boolean, optional, default=``False``)* – True for a deep network scan, looking for neighbors and their connections, False otherwise.

- **n_deep_scans** *(Integer, optional, default=1)* – Number of scans to perform before automatically stopping the discovery process. SCAN_TIL_CANCEL means the process will not be automatically stopped. Only applicable if deep=True.

**See also:**

- XBeeNetwork.add_device_discovered_callback()
- XBeeNetwork.add_discovery_process_finished_callback()
- XBeeNetwork.del_device_discovered_callback()
- XBeeNetwork.del_discovery_process_finished_callback()
- XBeeNetwork.get_deep_discovery_options()
- XBeeNetwork.set_deep_discovery_options()

**stop_discovery_process()**

Stops the discovery process if it is running.

Note that DigiMesh/DigiPoint devices are blocked until the discovery time configured (NT parameter) has elapsed, so if you try to get/set any parameter during the discovery process you will receive a timeout exception.

**class** digi.xbee.devices.NetworkEventType *(code, description)*

**Bases:** enum.Enum

Enumerates the different network event types.

**Values:**

- NetworkEventType.ADD = (0, ‘XBee added to the network’)
- NetworkEventType.DEL = (1, ‘XBee removed from the network’)
- NetworkEventType.UPDATE = (2, ‘XBee in the network updated’)
- NetworkEventType.CLEAR = (3, ‘Network cleared’)

**code**

Returns the code of the NetworkEventType element.

**Returns** the code of the NetworkEventType element.

**Return type** Integer

**description**

Returns the description of the NetworkEventType element.

**Returns** the description of the NetworkEventType element.
Return type  String
class  digi.xbee.devices.NetworkEventReason(code, description)
    Bases: enum.Enum
    Enumerates the different network event reasons.

Values:
    NetworkEventReason.DISCOVERED = (0, 'Discovered XBee')
    NetworkEventReason.NEIGHBOR = (1, 'Discovered as XBee neighbor')
    NetworkEventReason.RECEIVED_MSG = (2, 'Received message from XBee')
    NetworkEventReason.MANUAL = (3, 'Manual modification')
    NetworkEventReason.ROUTE = (4, 'Hop of a network route')
    NetworkEventReason.READ_INFO = (5, 'Read XBee information')

code
    Returns the code of the NetworkEventReason element.
    Returns the code of the NetworkEventReason element.
    Return type  Integer
description
    Returns the description of the NetworkEventReason element.
    Returns the description of the NetworkEventReason element.
    Return type  String
class  digi.xbee.devices.LinkQuality(lq=None, is_rssi=False)
    Bases: object
    This class represents the link quality of a connection. It can be a LQI (Link Quality Index) for ZigBee devices, or RSSI (Received Signal Strength Indicator) for the rest.
    Class constructor. Instanciates a new LinkQuality.
    Parameters
        •  lq (Integer, optional, default=``L_UNKNOWN``) – The link quality or None if unknown.
        •  is_rssi (Boolean, optional, default=``False``) – True to specify the value is a RSSI, False otherwise.
    UNKNOWN = <digi.xbee.devices.LinkQuality object>
    Unknown link quality.
    UNKNOWN_VALUE = -9999
    Unknown link quality value.

lq
    Returns the link quality value.
    Returns The link quality value.
    Return type  Integer

is_rssi
    Returns whether this is a RSSI value.
```
Returns True if this is an RSSI value, False for LQI.

Return type Boolean

class digi.xbee.devices.Connection(node_a, node_b, lq_a2b=None, lq_b2a=None, status_a2b=None, status_b2a=None)

Bases: object

This class represents a generic connection between two nodes in a XBee network. It contains the source and destination nodes, the LQI value for the connection between them and its status.

Class constructor. Instantiates a new Connection.

Parameters

- node_a (AbstractXBeeDevice) – One of the connection ends.
- node_b (AbstractXBeeDevice) – The other connection end.

- lq_a2b (LinkQuality or Integer, optional, default='None') – The link quality for the connection node_a -> node_b. If not specified LinkQuality.UNKNOWN is used.

- lq_b2a (LinkQuality or Integer, optional, default='None') – The link quality for the connection node_b -> node_a. If not specified LinkQuality.UNKNOWN is used.

- status_a2b (digi.xbee.models.zdo.RouteStatus, optional, default='None') – The status for the connection node_a -> node_b. If not specified RouteStatus.UNKNOWN is used.

- status_b2a (digi.xbee.models.zdo.RouteStatus, optional, default='None') – The status for the connection node_b -> node_a. If not specified RouteStatus.UNKNOWN is used.

Raises ValueError – if node_a or node_b is None.

See also:

AbstractXBeeDevice
LinkQuality
digi.xbee.models.zdo.RouteStatus

node_a

Returns the node A of this connection.

Returns The node A.

Return type AbstractXBeeDevice

See also:

AbstractXBeeDevice

node_b

Returns the node B of this connection.

Returns The node .

Return type AbstractXBeeDevice
```
See also:

AbstractXBeeDevice

$lq_{a2b}$
Returns the link quality of the connection from node A to node B.

- **Returns** The link quality for the connection A -> B.
- **Return type** LinkQuality

See also:

LinkQuality

$lq_{b2a}$
Returns the link quality of the connection from node B to node A.

- **Returns** The link quality for the connection B -> A.
- **Return type** LinkQuality

See also:

LinkQuality

$status_{a2b}$
Returns the status of this connection from node A to node B.

- **Returns** The status for A -> B connection.
- **Return type** digi.xbee.models.zdo.RouteStatus

See also:

digi.xbee.models.zdo.RouteStatus

$status_{b2a}$
Returns the status of this connection from node B to node A.

- **Returns** The status for B -> A connection.
- **Return type** digi.xbee.models.zdo.RouteStatus

See also:

digi.xbee.models.zdo.RouteStatus

$scan\_counter\_a2b$
Returns the scan counter for this connection, discovered by its A node.
Returns The scan counter for this connection, discovered by its A node.
Return type Integer

scan_counter_b2a
Returns the scan counter for this connection, discovered by its B node.
Returns The scan counter for this connection, discovered by its B node.
Return type Integer

digi.xbee.exception module

exception digi.xbee.exception.XBeeException
    Bases: Exception
    Generic XBee API exception. This class and its subclasses indicate conditions that an application might want to catch.
    All functionality of this class is the inherited of Exception.
    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.CommunicationException
    Bases: digi.xbee.exception.XBeeException
    This exception will be thrown when any problem related to the communication with the XBee device occurs.
    All functionality of this class is the inherited of Exception.
    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.ATCommandException
    (message='There was a problem sending the AT command packet.', cmd_status=None)
    Bases: digi.xbee.exception.CommunicationException
    This exception will be thrown when a response of a packet is not success or OK.
    All functionality of this class is the inherited of Exception.
    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.ConnectionException
    Bases: digi.xbee.exception.XBeeException
    This exception will be thrown when any problem related to the connection with the XBee device occurs.
    All functionality of this class is the inherited of Exception.
    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.XBeeDeviceException
    Bases: digi.xbee.exception.XBeeException
    This exception will be thrown when any problem related to the XBee device occurs.
    All functionality of this class is the inherited of Exception.
    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
exception digi.xbee.exception.InvalidConfigurationException
    (message='The configuration used to open the interface is invalid. ')

    Bases: digi.xbee.exception.ConnectionException

    This exception will be thrown when trying to open an interface with an invalid configuration.

    All functionality of this class is the inherited of Exception.

    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.InvalidOperatingModeException
    (message=None, op_mode=None)

    Bases: digi.xbee.exception.ConnectionException

    This exception will be thrown if the operating mode is different than OperatingMode.API_MODE and OperatingMode.API_MODE

    All functionality of this class is the inherited of Exception.

    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.InvalidPacketException
    (message='The XBee API packet is not properly formed. ')

    Bases: digi.xbee.exception.CommunicationException

    This exception will be thrown when there is an error parsing an API packet from the input stream.

    All functionality of this class is the inherited of Exception.

    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.OperationNotSupportedException
    (message='The requested operation is not supported by either the connection interface or the XBee device. ')

    Bases: digi.xbee.exception.XBeeDeviceException

    This exception will be thrown when the operation performed is not supported by the XBee device.

    All functionality of this class is the inherited of Exception.

    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.TimeoutException
    (message='There was a timeout while executing the requested operation. ')

    Bases: digi.xbee.exception.CommunicationException

    This exception will be thrown when performing synchronous operations and the configured time expires.

    All functionality of this class is the inherited of Exception.

    with_traceback()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
exception digi.xbee.exception.TransmitException (message='There was a problem with a transmitted packet response (status not ok)', transmit_status=None)

Bases: digi.xbee.exception.CommunicationException

This exception will be thrown when receiving a transmit status different than TransmitStatus.SUCCESS after sending an XBee API packet.

All functionality of this class is the inherited of Exception.

with_traceback()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.XBeeSocketException (message='There was a socket error', status=None)

Bases: digi.xbee.exception.XBeeException

This exception will be thrown when there is an error performing any socket operation.

All functionality of this class is the inherited of Exception.

with_traceback()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.FirmwareUpdateException

Bases: digi.xbee.exception.XBeeException

This exception will be thrown when any problem related to the firmware update process of the XBee device occurs.

All functionality of this class is the inherited of Exception.

with_traceback()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.RecoveryException

Bases: digi.xbee.exception.XBeeException

This exception will be thrown when any problem related to the auto-recovery process of the XBee device occurs.

All functionality of this class is the inherited of Exception.

with_traceback()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

digi.xbee.filesystem module

class digi.xbee.filesystem.FileSystemElement (name, path, size=0, is_directory=False)
    Bases: object

    Class used to represent XBee file system elements (files and directories).

    Class constructor. Instantiates a new FileSystemElement with the given parameters.

    Parameters
    • name (String) – the name of the file system element.
    • path (String) – the absolute path of the file system element.
    • size (Integer) – the size of the file system element, only applicable to files.
    • is_directory (Boolean) – True if the file system element is a directory, False if it is a file.
name
Returns the file system element name.

<table>
<thead>
<tr>
<th>Returns</th>
<th>the file system element name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return type</td>
<td>String</td>
</tr>
</tbody>
</table>

path
Returns the file system element absolute path.

<table>
<thead>
<tr>
<th>Returns</th>
<th>the file system element absolute path.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return type</td>
<td>String</td>
</tr>
</tbody>
</table>

size
Returns the file system element size.

| Returns       | the file system element size. If element is a directory, returns ‘0’.
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return type</td>
<td>Integer</td>
</tr>
</tbody>
</table>

is_directory
Returns whether the file system element is a directory or not.

<table>
<thead>
<tr>
<th>Returns</th>
<th>True if the file system element is a directory, False otherwise.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return type</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

is_secure
Returns whether the file system element is a secure element or not.

<table>
<thead>
<tr>
<th>Returns</th>
<th>True if the file system element is secure, False otherwise.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return type</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

exception
digi.xbee.filesystem.FileSystemException
Bases: digi.xbee.exception.XBeeException

This exception will be thrown when any problem related with the XBee device file system occurs.

All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception
digi.xbee.filesystem.FileSystemNotSupportedException
Bases: digi.xbee.filesystem.FileSystemException

This exception will be thrown when the file system feature is not supported in the device.

All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class
digi.xbee.filesystem.LocalXBeeFileSystemManager
(xbee_device)
Bases: object

Helper class used to manage the local XBee file system.

Class constructor. Instantiates a new LocalXBeeFileSystemManager with the given parameters.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>xbee_device (XBeeDevice) – the local XBee device to manage its file system.</th>
</tr>
</thead>
</table>

is_connected
Returns whether the file system manager is connected or not.

<table>
<thead>
<tr>
<th>Returns</th>
<th>True if the file system manager is connected, False otherwise.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>Return type</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
Return type  Boolean

c\textit{connect} ()
Connects the file system manager.

\textbf{Raises}

- \textit{FileSystemException} – if there is any error connecting the file system manager.
- \textit{FileSystemNotSupportedException} – if the device does not support filesystem feature.

\textit{disconnect} ()
Disconnects the file system manager and restores the device connection.

\textbf{Raises} \textit{XBeeException} – if there is any error restoring the XBee device connection.

\textit{get\_current\_directory} ()
Returns the current device directory.

\textbf{Returns} the current device directory.

\textbf{Return type}  String

\textbf{Raises} \textit{FileSystemException} – if there is any error getting the current directory or the function is not supported.

\textit{change\_directory} (directory)
Changes the current device working directory to the given one.

\textbf{Parameters} \textit{directory} (String) – the new directory to change to.

\textbf{Returns} the current device working directory after the directory change.

\textbf{Return type}  String

\textbf{Raises} \textit{FileSystemException} – if there is any error changing the current directory or the function is not supported.

\textit{make\_directory} (directory)
Creates the provided directory.

\textbf{Parameters} \textit{directory} (String) – the new directory to create.

\textbf{Raises} \textit{FileSystemException} – if there is any error creating the directory or the function is not supported.

\textit{list\_directory} (directory=\textit{None})
Lists the contents of the given directory.

\textbf{Parameters} \textit{directory} (String, \textit{optional}) – the directory to list its contents. Optional. If not provided, the current directory contents are listed.

\textbf{Returns} list of \textit{FilesystemElement} objects contained in the given (or current) directory.

\textbf{Return type}  List

\textbf{Raises} \textit{FileSystemException} – if there is any error listing the directory contents or the function is not supported.

\textit{remove\_element} (element\_path)
Removes the given file system element path.

\textbf{Parameters} \textit{element\_path} (String) – path of the file system element to remove.
Raises :class:`~xbee.exceptions.FileSystemException` – if there is any error removing the element or the function is not supported.

**move_element** (:paramref:`~xbasic.string source_path` :paramref:`~xbasic.string dest_path`)  
Moves the given source element to the given destination path.

**Parameters**

- **source_path** (:paramref:`~xbasic.string`) – source path of the element to move.
- **dest_path** (:paramref:`~xbasic.string`) – destination path of the element to move.

Raises :class:`~xbee.exceptions.FileSystemException` – if there is any error moving the element or the function is not supported.

**put_file** (:paramref:`~xbasic.string source_path` :paramref:`~xbasic.string dest_path`, :paramref:`~xbasic.boolean secure=False` :paramref:`~function progress_callback=None`)  
Transfers the given file in the specified destination path of the XBee device.

**Parameters**

- **source_path** (:paramref:`~xbasic.string`) – the path of the file to transfer.
- **dest_path** (:paramref:`~xbasic.string`) – the destination path to put the file in.
- **secure** (:paramref:`~xbasic.boolean`, optional) – True if the file should be stored securely, False otherwise. Defaults to False.
- **progress_callback** (:paramref:`~function`, optional) – function to execute to receive progress information.
  
  Takes the following arguments:
  
  - The progress percentage as integer.

Raises :class:`~xbee.exceptions.FileSystemException` – if there is any error transferring the file or the function is not supported.

**put_dir** (:paramref:`~xbasic.string source_dir` :paramref:`~xbasic.string dest_dir=None` :paramref:`~function progress_callback=None`)  
Uploads the given source directory contents into the given destination directory in the device.

**Parameters**

- **source_dir** (:paramref:`~xbasic.string`) – the local directory to upload its contents.
- **dest_dir** (:paramref:`~xbasic.string`, optional) – the remote directory to upload the contents to. Defaults to current directory.
- **progress_callback** (:paramref:`~function`, optional) – function to execute to receive progress information.

  Takes the following arguments:
  
  - The file being uploaded as string.
  - The progress percentage as integer.

Raises :class:`~xbee.exceptions.FileSystemException` – if there is any error uploading the directory or the function is not supported.

**get_file** (:paramref:`~xbasic.string source_path` :paramref:`~xbasic.string dest_path` :paramref:`~function progress_callback=None`)  
Downloads the given XBee device file in the specified destination path.

**Parameters**

- **source_path** (:paramref:`~xbasic.string`) – the path of the XBee device file to download.
- **dest_path** (:paramref:`~xbasic.string`) – the destination path to store the file in.
• progress_callback (Function, optional) – function to execute to receive progress information.

  Takes the following arguments:
  
  – The progress percentage as integer.

  Raises FileSystemException – if there is any error downloading the file or the function is not supported.

format filesystem ()

  Formats the device file system.

  Raises FileSystemException – if there is any error formatting the file system.

get usage information ()

  Returns the file system usage information.

  Returns collection of pair values describing the usage information.

  Return type Dictionary

  Raises FileSystemException – if there is any error retrieving the file system usage information.

get file hash (file_path)

  Returns the SHA256 hash of the given file path.

  Parameters file_path (String) – path of the file to get its hash.

  Returns the SHA256 hash of the given file path.

  Return type String

  Raises FileSystemException – if there is any error retrieving the file hash.

digi.xbee.firmware module

digi.xbee.firmware.update local firmware (target, xml firmware file, xbee firmware file=None, bootloader firmware file=None, timeout=None, progress callback=None)

  Performs a local firmware update operation in the given target.

  Parameters

  • target (String or XBeeDevice) – target of the firmware upload operation. String: serial port identifier. AbstractXBeeDevice: the XBee device to upload its firmware.

  • xml firmware file (String) – path of the XML file that describes the firmware to upload.

  • xbee firmware file (String, optional) – location of the XBee binary firmware file.

  • bootloader firmware file (String, optional) – location of the bootloader binary firmware file.

  • timeout (Integer, optional) – the serial port read data timeout.

  • progress callback (Function, optional) – function to execute to receive progress information. Receives two arguments:
digi.xbee.firmware.update_remote_firmware(remote_device, xml_firmware_file, ota_firmware_file=None, otb_firmware_file=None, timeout=None, progress_callback=None)

Performs a local firmware update operation in the given target.

Parameters

• remote_device (RemoteXBeeDevice) – remote XBee device to upload its firmware.

• xml_firmware_file (String) – path of the XML file that describes the firmware to upload.

• ota_firmware_file (String, optional) – path of the OTA firmware file to upload.

• otb_firmware_file (String, optional) – path of the OTB firmware file to upload (bootloader bundle).

• timeout (Integer, optional) – the timeout to wait for remote frame requests.

• progress_callback (Function, optional) – function to execute to receive progress information. Receives two arguments:
  – The current update task as a String
  – The current update task percentage as an Integer

Raises FirmwareUpdateException – if there is any error performing the remote firmware update.

digi.xbee.io module

class digi.xbee.io.IOLine (description, index, at_command, pwm_command=None)
  Bases: enum.Enum

Enumerates the different IO lines that can be found in the XBee devices.

Depending on the hardware and firmware of the device, the number of lines that can be used as well as their functionality may vary. Refer to the product manual to learn more about the IO lines of your XBee device.

Values:

IOLine.DIO0_AD0 = ('DIO0/AD0', 0, 'D0')
IOLine.DIO1_AD1 = ('DIO1/AD1', 1, 'D1')
IOLine.DIO2_AD2 = ('DIO2/AD2', 2, 'D2')
IOLine.DIO3_AD3 = ('DIO3/AD3', 3, 'D3')
IOLine.DIO4_AD4 = ('DIO4/AD4', 4, 'D4')
IOLine.DIO5_AD5 = ('DIO5/AD5', 5, 'D5')
IOLine.DIO6 = ('DIO6', 6, 'D6')
IOLine.DIO7 = ('DIO7', 7, 'D7')
IOLine.DIO8 = ('DIO8', 8, 'D8')
IOLine.DIO9 = ('DIO9', 9, 'D9')
IOLine.DIO10_PWM0 = ('DIO10/PWM0', 10, 'P0', 'M0')
IOLine.DIO11_PWM1 = ('DIO11/PWM1', 11, 'P1', 'M1')
IOLine.DIO12 = ('DIO12', 12, 'P2')
IOLine.DIO13 = ('DIO13', 13, 'P3')
IOLine.DIO14 = ('DIO14', 14, 'P4')
IOLine.DIO15 = ('DIO15', 15, 'P5')
IOLine.DIO16 = ('DIO16', 16, 'P6')
IOLine.DIO17 = ('DIO17', 17, 'P7')
IOLine.DIO18 = ('DIO18', 18, 'P8')
IOLine.DIO19 = ('DIO19', 19, 'P9')

has_pwm_capability()

Returns whether the IO line has PWM capability or not.

Returns True if the IO line has PWM capability, False otherwise.

Return type Boolean

description

String. The IO line description.

index

Integer. The IO line index.

at_command

String. The IO line AT command.

pwm_at_command

String. The IO line PWM AT command.

class digi.xbee.io.IOValue

Bases: enum.Enum

Enumerates the possible values of a IOLine configured as digital I/O.

Values:

IOValue.LOW = 4
IOValue.HIGH = 5

code

Integer. The IO value code.

class digi.xbee.io.IOSample

Bases: object

This class represents an IO Data Sample. The sample is built using the constructor. The sample contains an analog and digital mask indicating which IO lines are configured with that functionality.
Depending on the protocol the XBee device is executing, the digital and analog masks are retrieved in separated bytes (2 bytes for the digital mask and 1 for the analog mask) or merged contained (digital and analog masks are contained in 2 bytes).

Digital and analog channels masks Indicates which digital and ADC IO lines are configured in the module. Each bit corresponds to one digital or ADC IO line on the module:

```
bit 0 = DIO0
bit 1 = DIO1
bit 2 = DIO2
bit 3 = DIO3
bit 4 = DIO4
bit 5 = DIO5
bit 6 = DIO6
bit 7 = DIO7
bit 8 = DIO8
bit 9 = AD0
bit 10 = AD1
bit 11 = AD2
bit 12 = AD3
bit 13 = AD4
bit 14 = AD5
bit 15 = NA
```

Example: mask of 0x0C29 means DIO0, DIO3, DIO5, AD1 and AD2 enabled.
```
0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1
```

Digital Channel Mask Indicates which digital IO lines are configured in the module. Each bit corresponds to one digital IO line on the module:

```
bit 0 = DIO0AD0
bit 1 = DIO1AD1
bit 2 = DIO2AD2
bit 3 = DIO3AD3
bit 4 = DIO4AD4
bit 5 = DIO5AD5ASSOC
bit 6 = DIO6RTS
bit 7 = DIO7CTS
bit 8 = DIO8DTRSLLEEP_RQ
bit 9 = DIO9ON_SLEEP
bit 10 = DIO10PWM0RSSI
bit 11 = DIO11PWM1
bit 12 = DIO12CD
bit 13 = DIO13
bit 14 = DIO14
bit 15 = NA
```

Example: mask of 0x040B means DIO0, DIO1, DIO2, DIO3 and DIO10 enabled.
```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1
```

Analog Channel Mask Indicates which lines are configured as ADC. Each bit in the analog channel mask corresponds to one ADC line on the module.

```
bit 0 = AD0DIO0
bit 1 = AD1DIO1
bit 2 = AD2DIO2
bit 3 = AD3DIO3
bit 4 = AD4DIO4
```

(continues on next page)
bit 5 = ADSDIOSASSOC
bit 6 = NA
bit 7 = Supply Voltage Value

Example: mask of 0x83 means AD0, and AD1 enabled.

Class constructor. Instantiates a new `IOSample` object with the provided parameters.

Parameters `io_sample_payload` (`Bytearray`) – The payload corresponding to an IO sample.

Raises `ValueError` – if `io_sample_payload` length is less than 5.

static `min_io_sample_payload` ()

Returns the minimum IO sample payload length.

Returns the minimum IO sample payload length.

Return type `Integer`

has_digital_values ()

Checks whether the IOSample has digital values or not.

Returns True if the sample has digital values, False otherwise.

Return type `Boolean`

has_digital_value (`io_line`)  

Returns whether the IO sample contains a digital value for the provided IO line or not.

Parameters `io_line` (`IOLine`) – The IO line to check if it has a digital value.

Returns True if the given IO line has a digital value, False otherwise.

Return type `Boolean`

has_analog_value (`io_line`)  

Returns whether the given IOLine has an analog value or not.

Returns True if the given IOLine has an analog value, False otherwise.

Return type `Boolean`

has_analog_values ()

Returns whether the `IOSample` has analog values or not.

Returns Boolean. True if there are analog values, False otherwise.

has_power_supply_value ()

Returns whether the IOSample has power supply value or not.

Returns Boolean. True if the given IOLine has a power supply value, False otherwise.

get_digital_value (`io_line`)  

Returns the digital value of the provided IO line.

To verify if this sample contains a digital value for the given IOLine, use the method `IOSample.has_digital_value()`.

Parameters `io_line` (`IOLine`) – The IO line to get its digital value.

Returns

The `IOValue` of the given IO line or `None` if the IO sample does not contain a digital value for the given IO line.
get_analog_value(io_line)
Returns the analog value of the provided IO line.
To verify if this sample contains an analog value for the given IOLine, use the method IOSample. has_analog_value().

Parameters io_line (IOLine) – The IO line to get its analog value.

Returns
The analog value of the given IO line or None if the IO sample does not contain an analog value for the given IO line.

Return type Integer

See also: IOLine

---

digital_hsb_mask
Integer. High Significant Byte (HSB) of the digital mask.

digital_lsb_mask
Integer. Low Significant Byte (LSB) of the digital mask.

digital_mask
Integer. Digital mask of the IO sample.

analog_mask
Integer. Analog mask of the IO sample.

digital_values
Dictionary. Digital values map.

analog_values
Dictionary. Analog values map.

power_supply_value
Integer. Power supply value, None if the sample does not contain power supply value.

class digi.xbee.io.IOMode
Bases: enum.Enum

Enumerates the different Input/Output modes that an IO line can be configured with.

DISABLED = 0
Disabled

SPECIAL_FUNCTIONALITY = 1
Firmware special functionality
PWM = 2
   PWM output

ADC = 2
   Analog to Digital Converter

DIGITAL_IN = 3
   Digital input

DIGITAL_OUT_LOW = 4
   Digital output, Low

DIGITAL_OUT_HIGH = 5
   Digital output, High

I2C_FUNCTIONALITY = 6
   I2C functionality

digi.xbee.profile module

class digi.xbee.profile.FirmwareBaudrate(index, baudrate)
Bases: enum.Enum
   This class lists the available firmware baudrate options for XBee Profiles.

Inherited properties:
   name (String): The name of this FirmwareBaudrate.
   value (Integer): The ID of this FirmwareBaudrate.

Values:
   FirmwareBaudrate.BD_1200 = (0, 1200)
   FirmwareBaudrate.BD_2400 = (1, 2400)
   FirmwareBaudrate.BD_4800 = (2, 4800)
   FirmwareBaudrate.BD_9600 = (3, 9600)
   FirmwareBaudrate.BD_19200 = (4, 19200)
   FirmwareBaudrate.BD_38400 = (5, 38400)
   FirmwareBaudrate.BD_57600 = (6, 57600)
   FirmwareBaudrate.BD_115200 = (7, 115200)
   FirmwareBaudrate.BD_230400 = (8, 230400)
   FirmwareBaudrate.BD_460800 = (9, 460800)
   FirmwareBaudrate.BD_921600 = (10, 921600)

   index
      Returns the index of the FirmwareBaudrate element.
      Returns the index of the FirmwareBaudrate element.
      Return type Integer

   baudrate
      Returns the baudrate of the FirmwareBaudrate element.
Returns the baudrate of the FirmwareBaudrate element.

**Return type** Integer

```python
class digi.xbee.profile.FirmwareParity(index, parity)
```

This class lists the available firmware parity options for XBee Profiles.

Inherited properties:

- **name** (String): The name of this FirmwareParity.
- **value** (Integer): The ID of this FirmwareParity.

Values:

- **FirmwareParity.NONE** = (0, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd587ca3f98>)
- **FirmwareParity.EVEN** = (1, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd587367400>)
- **FirmwareParity.ODD** = (2, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd5879dec18>)
- **FirmwareParity.MARK** = (3, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd5879dec50>)
- **FirmwareParity.SPACE** = (4, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd5879dec88>)

```python
index
```

Returns the index of the FirmwareParity element.

**Returns** the index of the FirmwareParity element.

**Return type** Integer

```python
parity
```

Returns the parity of the FirmwareParity element.

**Returns** the parity of the FirmwareParity element.

**Return type** String

```python
class digi.xbee.profile.FirmwareStopbits(index, stop_bits)
```

This class lists the available firmware stop bits options for XBee Profiles.

Inherited properties:

- **name** (String): The name of this FirmwareStopbits.
- **value** (Integer): The ID of this FirmwareStopbits.

Values:

- **FirmwareStopbits.SB_1** = (0, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd587f94a90>)
- **FirmwareStopbits.SB_2** = (1, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd5879decf8>)
- **FirmwareStopbits.SB_1_5** = (2, <sphinx.ext.autodoc.importer._MockObject object at 0x7fd58779f0b8>)
index
Returns the index of the FirmwareStopbits element.

Returns the index of the FirmwareStopbits element.

Return type Integer

stop_bits
Returns the stop bits of the FirmwareStopbits element.

Returns the stop bits of the FirmwareStopbits element.

Return type Float

class digi.xbee.profile.FlashFirmwareOption (code, description)
Bases: enum.Enum
This class lists the available flash firmware options for XBee Profiles.

Inherited properties:

name (String): The name of this FlashFirmwareOption.

value (Integer): The ID of this FlashFirmwareOption.

Values:
FlashFirmwareOption.FLASH_ALWAYS = (0, ‘Flash always’)
FlashFirmwareOption.FLASH_DIFFERENT = (1, ‘Flash firmware if it is different’)
FlashFirmwareOption.DONT_FLASH = (2, ‘Do not flash firmware’)

code
Returns the code of the FlashFirmwareOption element.

Returns the code of the FlashFirmwareOption element.

Return type Integer
description
Returns the description of the FlashFirmwareOption element.

Returns the description of the FlashFirmwareOption element.

Return type String
class digi.xbee.profile.XBeeSettingType (tag, description)
Bases: enum.Enum
This class lists the available firmware setting types.

Inherited properties:

name (String): The name of this XBeeSettingType.

value (Integer): The ID of this XBeeSettingType.

Values:
XBeeSettingType.NUMBER = (‘number’, ‘Number’)
XBeeSettingType.COMBO = (‘combo’, ‘Combo’)

2.5. API reference
XBeeSettingType.TEXT = ('text', 'Text')
XBeeSettingType.BUTTON = ('button', 'Button')
XBeeSettingType.NO_TYPE = ('none', 'No type')

tag
Returns the tag of the XBeeSettingType element.

    Returns the tag of the XBeeSettingType element.
    Return type String
description
Returns the description of the XBeeSettingType element.

    Returns the description of the XBeeSettingType element.
    Return type String
class digi.xbee.profile.XBeeSettingFormat (tag, description)
Bases: enum.Enum
    This class lists the available text firmware setting formats.

Inherited properties:
    name (String): The name of this XBeeSettingFormat.
    value (Integer): The ID of this XBeeSettingFormat.

Values:
    XBeeSettingFormat.HEX = ('HEX', 'Hexadecimal')
    XBeeSettingFormat.ASCII = ('ASCII', 'ASCII')
    XBeeSettingFormat.IPV4 = ('IPV4', 'IPv4')
    XBeeSettingFormat.IPV6 = ('IPV6', 'IPv6')
    XBeeSettingFormat.PHONE = ('PHONE', 'phone')
    XBeeSettingFormat.NO_FORMAT = ('none', 'No format')

tag
Returns the tag of the XBeeSettingFormat element.

    Returns the tag of the XBeeSettingFormat element.
    Return type String
description
Returns the description of the XBeeSettingFormat element.

    Returns the description of the XBeeSettingFormat element.
    Return type String
class digi.xbee.profile.XBeeProfileSetting (name, setting_type, setting_format, value)
Bases: object
    This class represents an XBee profile setting and provides information like the setting name, type, format and value.
Class constructor. Instantiates a new XBeeProfileSetting with the given parameters.

Parameters
- **name** (*String*) – the setting name
- **setting_type** (*XBeeSettingType*) – the setting type
- **setting_format** (*XBeeSettingType*) – the setting format
- **value** (*String*) – the setting value

**name**
Returns the XBee setting name.

Return type  *String*

**type**
Returns the XBee setting type.

Return type  *XBeeSettingType*

**format**
Returns the XBee setting format.

Return type  *XBeeSettingFormat*

**value**
Returns the XBee setting value as string.

Return type  *String*

**bytearray_value**
Returns the XBee setting value as bytearray to be set in the device.

Return type  *Bytearray*

---

**exception**  *digi.xbee.profile.ReadProfileException*
**Bases:**  *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem reading the XBee profile occurs.

All functionality of this class is the inherited from *Exception*.

**with_traceback()**
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

**exception**  *digi.xbee.profile.UpdateProfileException*
**Bases:**  *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem updating the XBee profile into a device occurs.

All functionality of this class is the inherited from *Exception*.

**with_traceback()**
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
class digi.xbee.profile.XBeeProfile(profile_file)
    Bases: object

Helper class used to manage serial port break line in a parallel thread.

Class constructor. Instantiates a new XBeeProfile with the given parameters.

Parameters profile_file (String) – path of the '.xpro' profile file.

Raises

* ProfileReadException – if there is any error reading the profile file.
* ValueError – if the provided profile file is not valid

get_setting_default_value(setting_name)
Returns the default value of the given firmware setting.

Parameters setting_name (String) – the name of the setting to retrieve its default value.

Returns the default value of the setting, None if the setting is not found or it has no default value.

Return type String

profile_file
Returns the profile file.

Returns the profile file.

Return type String

version
Returns the profile version.

Returns the profile version.

Return type String

flash_firmware_option
Returns the profile flash firmware option.

Returns the profile flash firmware option.

Return type FlashFirmwareOption

See also:

FlashFirmwareOption

description
Returns the profile description.

Returns the profile description.

Return type String

reset_settings
Returns whether the settings of the XBee device will be reset before applying the profile ones or not.

Returns

True if the settings of the XBee device will be reset before applying the profile ones, False otherwise.
Return type  Boolean

**has_filesystem**
Returns whether the profile has filesystem information or not.

**Returns**  True if the profile has filesystem information, False otherwise.

**Return type**  Boolean

**profile_settings**
Returns all the firmware settings that the profile configures.

**Returns**  a list with all the firmware settings that the profile configures

(XBeeProfileSetting).

**Return type**  List

**firmware_version**
Returns the compatible firmware version of the profile.

**Returns**  the compatible firmware version of the profile.

**Return type**  Integer

**hardware_version**
Returns the compatible hardware version of the profile.

**Returns**  the compatible hardware version of the profile.

**Return type**  Integer

**firmware_description_file**
Returns the path of the profile firmware description file.

**Returns**  the path of the profile firmware description file.

**Return type**  String

**file_system_path**
Returns the profile file system path.

**Returns**  the path of the profile file system directory.

**Return type**  String

**digi.xbee.profile.apply_xbee_profile**(xbee_device, profile_path, progress_callback=None)
Applies the given XBee profile into the given XBee device.

**Parameters**

- **xbee_device** (XBeeDevice or RemoteXBeeDevice) – the XBee device to apply profile to.
- **profile_path** (String) – path of the XBee profile file to apply.
- **progress_callback** (Function, optional) – function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

**Raises**

- **ValueError** – if the XBee profile or the XBee device is not valid.
• *UpdateProfileException* – if there is any error during the update XBee profile operation.

**digi.xbee.reader module**

```python
class digi.xbee.reader.XBeeEvent
    Bases: list

    This class represents a generic XBee event.
    New event callbacks can be added here following this prototype:

    ```python
def callback_prototype(*args, **kwargs):
        #do something...
    ```

    All of them will be executed when the event is fired.

    **See also:**

    list (Python standard class)

    ```python
    def append():
        Append object to the end of the list.

def clear():
    Remove all items from list.

def copy():
    Return a shallow copy of the list.

def count():
    Return number of occurrences of value.

def extend():
    Extend list by appending elements from the iterable.

def index():
    Return first index of value.
    Raises ValueError if the value is not present.

def insert():
    Insert object before index.

def pop():
    Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

def remove():
    Remove first occurrence of value.
    Raises ValueError if the value is not present.

def reverse():
    Reverse IN PLACE.

def sort():
    Stable sort IN PLACE.
```
```
class digi.xbee.reader.PacketReceived
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives any packet, independent of its frame type.
The callbacks for handle this events will receive the following arguments:

1. received_packet (XBeeAPIPacket): the received packet.

See also:

XBeeAPIPacket
XBeeEvent

append()
    Append object to the end of the list.

clear()
    Remove all items from list.

copy()
    Return a shallow copy of the list.

count()
    Return number of occurrences of value.

extend()
    Extend list by appending elements from the iterable.

index()
    Return first index of value.
    Raises ValueError if the value is not present.

insert()
    Insert object before index.

pop()
    Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

remove()
    Remove first occurrence of value.
    Raises ValueError if the value is not present.

reverse()
    Reverse IN PLACE.

sort()
    Stable sort IN PLACE.

class digi.xbee.reader.DataReceived
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives data.
The callbacks for handle this events will receive the following arguments:

1. message (XBeeMessage): message containing the data received, the sender and the time.

See also:

2.5. API reference
XBeeEvent
XBeeMessage

append()
Append object to the end of the list.
clear()
Remove all items from list.
copy()
Return a shallow copy of the list.
count()
Return number of occurrences of value.
extend()
Extend list by appending elements from the iterable.
index()
Return first index of value.
    Raises ValueError if the value is not present.
insert()
Insert object before index.
pop()
Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.
remove()
Remove first occurrence of value.
    Raises ValueError if the value is not present.
reverse()
Reverse IN PLACE.
sort()
Stable sort IN PLACE.
class digi.xbee.reader.ModemStatusReceived
Bases: digi.xbee.reader.XBeeEvent
This event is fired when a XBee receives a modem status packet.
The callbacks for handle this events will receive the following arguments:
    1. modem_status (ModemStatus): the modem status received.
See also:

XBeeEvent
ModemStatus

append()
Append object to the end of the list.
clear()
Remove all items from list.
copy ()
Return a shallow copy of the list.

count ()
Return number of occurrences of value.

extend ()
Extend list by appending elements from the iterable.

index ()
Return first index of value.
Raising ValueError if the value is not present.

insert ()
Insert object before index.

pop ()
Remove and return item at index (default last).
Raising IndexError if list is empty or index is out of range.

remove ()
Remove first occurrence of value.
Raising ValueError if the value is not present.

reverse ()
Reverse IN PLACE.

sort ()
Stable sort IN PLACE.

class digi.xbee.reader.IOSampleReceived
Bases: digi.xbee.reader.XBeeEvent

This event is fired when a XBee receives an IO packet.

This includes:
1. IO data sample RX indicator packet.
2. RX IO 16 packet.
3. RX IO 64 packet.

The callbacks that handle this event will receive the following arguments:
1. io_sample (IOSample): the received IO sample.
2. sender (RemoteXBeeDevice): the remote XBee device who has sent the packet.
3. time (Integer): the time in which the packet was received.

See also:

IOSample
RemoteXBeeDevice
XBeeEvent

append ()
Append object to the end of the list.

clear ()
Remove all items from list.
copy()  
Return a shallow copy of the list.

count()  
Return number of occurrences of value.

extend()  
Extend list by appending elements from the iterable.

index()  
Return first index of value.  
 Raises ValueError if the value is not present.

insert()  
Insert object before index.

pop()  
Remove and return item at index (default last).  
 Raises IndexError if list is empty or index is out of range.

remove()  
Remove first occurrence of value.  
 Raises ValueError if the value is not present.

reverse()  
Reverse IN PLACE.

sort()  
Stable sort IN PLACE.

class digi.xbee.reader.NetworkModified  
Bases: digi.xbee.reader.XBeeEvent

This event is fired when the network is being modified by the addition of a new node, an existing node information is updated, a node removal, or when the network items are cleared.  
The callbacks that handle this event will receive the following arguments:

1. event_type (digi.xbee.devices.NetworkEventType): the network event type.
3. node (digi.xbee.devices.XBeeDevice or digi.xbee.devices.RemoteXBeeDevice): The node added, updated or removed from the network.

See also:

digi.xbee.devices.NetworkEventReason  
digi.xbee.devices.NetworkEventType  
digi.xbee.devices.RemoteXBeeDevice  
digi.xbee.devices.XBeeDevice
XBeeEvent

append()  
Append object to the end of the list.

clear()  
Remove all items from list.
copy()  
Return a shallow copy of the list.

count()  
Return number of occurrences of value.

extend()  
Extend list by appending elements from the iterable.

index()  
Return first index of value.
  Raises ValueError if the value is not present.

insert()  
Insert object before index.

pop()  
Remove and return item at index (default last).
  Raises IndexError if list is empty or index is out of range.

remove()  
Remove first occurrence of value.
  Raises ValueError if the value is not present.

reverse()  
Reverse IN PLACE.

sort()  
Stable sort IN PLACE.

class digi.xbee.reader.DeviceDiscovered  
Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee discovers another remote XBee during a discovering operation.  
The callbacks that handle this event will receive the following arguments:

  1. discovered_device (RemoteXBeeDevice): the discovered remote XBee device.

See also:

RemoteXBeeDevice
XBeeEvent

append()  
Append object to the end of the list.

clear()  
Remove all items from list.

copy()  
Return a shallow copy of the list.

count()  
Return number of occurrences of value.

extend()  
Extend list by appending elements from the iterable.
index()
    Return first index of value.
    Raises ValueError if the value is not present.

insert()
    Insert object before index.

pop()
    Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

remove()
    Remove first occurrence of value.
    Raises ValueError if the value is not present.

reverse()
    Reverse IN PLACE.

sort()
    Stable sort IN PLACE.

class digi.xbee.reader.DiscoveryProcessFinished
    Bases: digi.xbee.reader.XBeeEvent

    This event is fired when the discovery process finishes, either successfully or due to an error.
    The callbacks that handle this event will receive the following arguments:

    1. status (NetworkDiscoveryStatus): the network discovery status.

    See also:

    NetworkDiscoveryStatus
    XBeeEvent

append()
    Append object to the end of the list.

clear()
    Remove all items from list.

copy()
    Return a shallow copy of the list.

count()
    Return number of occurrences of value.

extend()
    Extend list by appending elements from the iterable.

index()
    Return first index of value.
    Raises ValueError if the value is not present.

insert()
    Insert object before index.

pop()
    Remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

**remove()**
Remove first occurrence of value.
Raises ValueError if the value is not present.

**reverse()**
Reverse **IN PLACE**.

**sort()**
Stable sort **IN PLACE**.

### class digi.xbee.reader.ExplicitDataReceived
digimod.exe.reader.XBeeEvent

This event is fired when an XBee receives an explicit data packet.
The callbacks for handle this events will receive the following arguments:

1. **message (ExplicitXBeeMessage):** message containing the data received, the sender, the time and explicit data message parameters.

See also:

**XBeeEvent**
**XBeeMessage**

**append()**
Append object to the end of the list.

**clear()**
Remove all items from list.

**copy()**
Return a shallow copy of the list.

**count()**
Return number of occurrences of value.

**extend()**
Extend list by appending elements from the iterable.

**index()**
Return first index of value.
Raises ValueError if the value is not present.

**insert()**
Insert object before index.

**pop()**
Remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

**remove()**
Remove first occurrence of value.
Raises ValueError if the value is not present.

**reverse()**
Reverse **IN PLACE**.
sort()
    Stable sort IN PLACE.

class digi.xbee.reader.IPDataReceived
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives IP data.
The callbacks for handle this events will receive the following arguments:

1. message (IPMessage): message containing containing the IP address the message belongs to, the source and destination ports, the IP protocol, and the content (data) of the message.

See also:

XBeeEvent
IPMessage

append()
    Append object to the end of the list.

clear()
    Remove all items from list.

copy()
    Return a shallow copy of the list.

count()
    Return number of occurrences of value.

extend()
    Extend list by appending elements from the iterable.

index()
    Return first index of value.
    Raises ValueError if the value is not present.

insert()
    Insert object before index.

pop()
    Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

remove()
    Remove first occurrence of value.
    Raises ValueError if the value is not present.

reverse()
    Reverse IN PLACE.

sort()
    Stable sort IN PLACE.

class digi.xbee.reader.SMSReceived
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives an SMS.
The callbacks for handle this events will receive the following arguments:
1. **message** (*SMSSMessage*): message containing the phone number that sent the message and the content (data) of the message.

   See also:

   - XBeeEvent
   - SMSSMessage

append()  
  Append object to the end of the list.

clear()  
  Remove all items from list.

copy()  
  Return a shallow copy of the list.

count()  
  Return number of occurrences of value.

extend()  
  Extend list by appending elements from the iterable.

index()  
  Return first index of value.
  Raises ValueError if the value is not present.

insert()  
  Insert object before index.

pop()  
  Remove and return item at index (default last).
  Raises IndexError if list is empty or index is out of range.

remove()  
  Remove first occurrence of value.
  Raises ValueError if the value is not present.

reverse()  
  Reverse IN PLACE.

sort()  
  Stable sort IN PLACE.

class digi.xbee.reader.RelayDataReceived  
  Bases: digi.xbee.reader.XBeeEvent

  This event is fired when an XBee receives a user data relay output packet.
  The callbacks to handle these events will receive the following arguments:

  1. **message** (*UserDataRelayMessage*): message containing the source interface and the content (data) of the message.

  See also:

  - XBeeEvent
  - UserDataRelayMessage

2.5. API reference 751
append()
   Append object to the end of the list.

clear()
   Remove all items from list.

copy()
   Return a shallow copy of the list.

count()
   Return number of occurrences of value.

extend()
   Extend list by appending elements from the iterable.

index()
   Return first index of value.
   Raises ValueError if the value is not present.

insert()
   Insert object before index.

pop()
   Remove and return item at index (default last).
   Raises IndexError if list is empty or index is out of range.

remove()
   Remove first occurrence of value.
   Raises ValueError if the value is not present.

reverse()
   Reverse IN PLACE.

sort()
   Stable sort IN PLACE.

class digi.xbee.reader.BluetoothDataReceived
Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives data from the Bluetooth interface.

The callbacks to handle these events will receive the following arguments:
   1. data (Bytearray): received Bluetooth data.

See also:

XBeeEvent
extend()  
Extend list by appending elements from the iterable.

index()  
Return first index of value.  
Raises ValueError if the value is not present.

insert()  
Insert object before index.

pop()  
Remove and return item at index (default last).  
Raises IndexError if list is empty or index is out of range.

remove()  
Remove first occurrence of value.  
Raises ValueError if the value is not present.

reverse()  
Reverse IN PLACE.

sort()  
Stable sort IN PLACE.

class digi.xbee.reader.MicroPythonDataReceived
Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives data from the MicroPython interface.  
The callbacks to handle these events will receive the following arguments:

1. data (Bytearray): received MicroPython data.

See also:

XBeeEvent

append()  
Append object to the end of the list.

clear()  
Remove all items from list.

copy()  
Return a shallow copy of the list.

count()  
Return number of occurrences of value.

extend()  
Extend list by appending elements from the iterable.

index()  
Return first index of value.  
Raises ValueError if the value is not present.

insert()  
Insert object before index.
pop()
Remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove()
Remove first occurrence of value.
Raises ValueError if the value is not present.

reverse()
Reverse IN PLACE.

sort()
Stable sort IN PLACE.

class digi.xbee.reader.SocketStateReceived
Bases: digi.xbee.reader.XBeeEvent
This event is fired when an XBee receives a socket state packet.
The callbacks to handle these events will receive the following arguments:

1. socket_id (Integer): socket ID for state reported.
2. state (SocketState): received state.
See also:

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.
Raises ValueError if the value is not present.

insert()
Insert object before index.

pop()
Remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove()
Remove first occurrence of value.
Raises ValueError if the value is not present.
reverse()
    Reverse IN PLACE.

sort()
    Stable sort IN PLACE.

class digi.xbee.reader.SocketDataReceived
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives a socket receive data packet.

The callbacks to handle these events will receive the following arguments:

1. socket_id (Integer): ID of the socket that received the data.
2. payload (Bytearray): received data.

See also:

XBeeEvent

append()
    Append object to the end of the list.

clear()
    Remove all items from list.

copy()
    Return a shallow copy of the list.

count()
    Return number of occurrences of value.

extend()
    Extend list by appending elements from the iterable.

index()
    Return first index of value.
    Raises ValueError if the value is not present.

insert()
    Insert object before index.

pop()
    Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

remove()
    Remove first occurrence of value.
    Raises ValueError if the value is not present.

reverse()
    Reverse IN PLACE.

sort()
    Stable sort IN PLACE.

class digi.xbee.reader.SocketDataReceivedFrom
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when an XBee receives a socket receive from data packet.

2.5. API reference 755
The callbacks to handle these events will receive the following arguments:

1. socket_id (Integer): ID of the socket that received the data.

2. address (Tuple): a pair (host, port) of the source address where host is a string representing an IPv4 address like '100.50.200.5', and port is an integer.

3. payload (Bytearray): received data.

See also:

```
XBeeEvent
```

```
append()
    Append object to the end of the list.

clear()
    Remove all items from list.

copy()
    Return a shallow copy of the list.

count()
    Return number of occurrences of value.

extend()
    Extend list by appending elements from the iterable.

index()
    Return first index of value.
    Raises ValueError if the value is not present.

insert()
    Insert object before index.

pop()
    Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

remove()
    Remove first occurrence of value.
    Raises ValueError if the value is not present.

reverse()
    Reverse IN PLACE.

sort()
    Stable sort IN PLACE.
```

class digi.xbee.reader.RouteRecordIndicatorReceived
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when a route record packet is received.

The callbacks to handle these events will receive the following arguments:

1. Source (RemoteXBeeDevice): The remote node that sent the route record.

2. Hops (List): List of intermediate hops 16-bit addresses from closest to source (who sent the route record) to closest to destination (XBeel6BitAddress).

See also:
XBeeEvent

**append()**
Append object to the end of the list.

**clear()**
Remove all items from list.

**copy()**
Return a shallow copy of the list.

**count()**
Return number of occurrences of value.

**extend()**
Extend list by appending elements from the iterable.

**index()**
Return first index of value.
Raises ValueError if the value is not present.

**insert()**
Insert object before index.

**pop()**
Remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

**remove()**
Remove first occurrence of value.
Raises ValueError if the value is not present.

**reverse()**
Reverse *IN PLACE*.

**sort()**
Stable sort *IN PLACE*.

---

**class** digi.xbee.reader.RouteInformationReceived

**Bases:** digi.xbee.reader.XBeeEvent

This event is fired when a route information packet is received.

**The callbacks to handle these events will receive the following arguments:**

1. Source event (Integer): The source event (0x11: NACK, 0x12: Trace route)
2. **Timestamp (Integer): The system timer value on the node generating** this package. The timestamp is in microseconds.
3. ACK timeout count (Integer): Number of MAC ACK timeouts that occur.
4. **TX blocked count (Integer): Number of times the transmissions was** blocked due to reception in progress.
5. **Destination address** (**XBee64BitAddress**): 64-bit address of the final destination node.
6. **Source address** (**XBee64BitAddress**): 64-bit address of the source node.
7. **Responder address** (**XBee64BitAddress**): 64-bit address of the node that generates this packet after it sends (or attempts to send) the packet to the next hop (successor node)
8. **Successor address (XBee64BitAddress):** 64-bit address of the next node after the responder in the route towards the destination.

**See also:**

**XBeeEvent**

```python
append()  # Append object to the end of the list.
clear()  # Remove all items from list.
copy()  # Return a shallow copy of the list.
count()  # Return number of occurrences of value.
extend()  # Extend list by appending elements from the iterable.
index()  # Return first index of value.
        # Raises ValueError if the value is not present.
insert()  # Insert object before index.
pop()  # Remove and return item at index (default last).
        # Raises IndexError if list is empty or index is out of range.
remove()  # Remove first occurrence of value.
        # Raises ValueError if the value is not present.
reverse()  # Reverse IN PLACE.
sort()  # Stable sort IN PLACE.
```

class digi.xbee.reader.RouteReceived
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when a route is received.

**The callbacks to handle these events will receive the following arguments:**

1. **source (XBeeDevice):** The local node.
2. **destination (RemoteXBeeDevice):** The remote node.
3. **hops (List):** List of intermediate hops from source node to closest to destination (RemoteXBeeDevice).

**See also:**

**XBeeEvent**
append()  
Append object to the end of the list.

clear()  
Remove all items from list.

copy()  
Return a shallow copy of the list.

count()  
Return number of occurrences of value.

extend()  
Extend list by appending elements from the iterable.

index()  
Return first index of value.  
Raises ValueError if the value is not present.

insert()  
Insert object before index.

pop()  
Remove and return item at index (default last).  
Raises IndexError if list is empty or index is out of range.

remove()  
Remove first occurrence of value.  
Raises ValueError if the value is not present.

reverse()  
Reverse IN PLACE.

sort()  
Stable sort IN PLACE.

class digi.xbee.reader.InitDiscoveryScan
Bases: digi.xbee.reader.XBeeEvent

This event is fired when a new network discovery scan is about to start.  
The callbacks to handle these events will receive the following arguments:

1. Number of scan to start (starting with 1).
2. Total number of scans.

See also:

XBeeEvent

append()  
Append object to the end of the list.

clear()  
Remove all items from list.

copy()  
Return a shallow copy of the list.
count()
    Return number of occurrences of value.

extend()
    Extend list by appending elements from the iterable.

index()
    Return first index of value.
    Raises ValueError if the value is not present.

insert()
    Insert object before index.

pop()
    Remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

remove()
    Remove first occurrence of value.
    Raises ValueError if the value is not present.

reverse()
    Reverse IN PLACE.

sort()
    Stable sort IN PLACE.

class digi.xbee.reader.EndDiscoveryScan
    Bases: digi.xbee.reader.XBeeEvent

This event is fired when a network discovery scan has just finished.
The callbacks to handle these events will receive the following arguments:

1. Number of scan that has finished (starting with 1).
2. Total number of scans.

See also:

XBeeEvent

append()
    Append object to the end of the list.

clear()
    Remove all items from list.

copy()
    Return a shallow copy of the list.

count()
    Return number of occurrences of value.

extend()
    Extend list by appending elements from the iterable.

index()
    Return first index of value.
    Raises ValueError if the value is not present.
**insert()**
Insert object before index.

**pop()**
Remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

**remove()**
Remove first occurrence of value.
Raises ValueError if the value is not present.

**reverse()**
Reverse IN PLACE.

**sort()**
Stable sort IN PLACE.

```python
class digi.xbee.reader.PacketListener (comm_iface, xbee_device, queue_max_size=None)
Bases: threading.Thread
This class represents a packet listener, which is a thread that’s always listening for incoming packets to the XBee.

When it receives a packet, this class throws an event depending on which packet it is. You can add your own callbacks for this events via certain class methods. This callbacks must have a certain header, see each event documentation.

This class has fields that are events. Its recommended to use only the append() and remove() method on them, or -= and += operators. If you do something more with them, it’s for your own risk.

Here are the parameters which will be received by the event callbacks, depending on which event it is in each case:

The following parameters are passed via **kwargs to event callbacks of:

1. **PacketReceived**
   1.1 received_packet (XBeeAPIPacket): the received packet.
2. **DataReceived**
   2.1 message (XBeeMessage): message containing the data received, the sender and the time.
3. **ModemStatusReceived**
   3.1 modem_status (ModemStatus): the modem status received.

Class constructor. Instantiates a new PacketListener object with the provided parameters.

**Parameters**

- **comm_iface** (XBeeCommunicationInterface) – the hardware interface to listen to.
- **xbee_device** (XBeeDevice) – the XBee that is the listener owner.
- **queue_max_size** (Integer) – the maximum size of the XBee queue.

**daemon**
A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

**wait_until_started** (timeout=None)
Blocks until the thread has fully started. If already started, returns immediately.

**Parameters**

- **timeout** (Float) – timeout for the operation in seconds.
run()
This is the method that will be executing for listening packets.

For each packet, it will execute the proper callbacks.

stop()
Stops listening.

is_running()
Returns whether this instance is running or not.

Returns True if this instance is running, False otherwise.

Return type Boolean

get_queue()
Returns the packets queue.

Returns the packets queue.

Return type XBeeQueue

get_data_queue()
Returns the data packets queue.

Returns the data packets queue.

Return type XBeeQueue

get_explicit_queue()
Returns the explicit packets queue.

Returns the explicit packets queue.

Return type XBeeQueue

get_ip_queue()
Returns the IP packets queue.

Returns the IP packets queue.

Return type XBeeQueue

add_packet_received_callback (callback)
Adds a callback for the event PacketReceived.

Parameters callback (Function or List of functions) – the callback. Receives two arguments.

• The received packet as a XBeeAPIPacket

add_data_received_callback (callback)
Adds a callback for the event DataReceived.

Parameters callback (Function or List of functions) – the callback. Receives one argument.

• The data received as an XBeeMessage

add_modem_status_received_callback (callback)
Adds a callback for the event ModemStatusReceived.

Parameters callback (Function or List of functions) – the callback. Receives one argument.

• The modem status as a ModemStatus
add_io_sample_received_callback(callback)
   Adds a callback for the event IOSampleReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives three arguments.
   • The received IO sample as an IOSample
   • The remote XBee device who has sent the packet as a RemoteXBeeDevice
   • The time in which the packet was received as an Integer

add_explicit_data_received_callback(callback)
   Adds a callback for the event ExplicitDataReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives one argument.
   • The explicit data received as an ExplicitXBeeMessage

add_ip_data_received_callback(callback)
   Adds a callback for the event IPDataReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives one argument.
   • The data received as an IPMessage

add_sms_received_callback(callback)
   Adds a callback for the event SMSReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives one argument.
   • The data received as an SMSMessage

add_user_data_relay_received_callback(callback)
   Adds a callback for the event RelayDataReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives one argument.
   • The data received as a UserDataRelayMessage

add.bluetooth_data_received_callback(callback)
   Adds a callback for the event BluetoothDataReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives one argument.
   • The data received as a Bytearray

add микропитание_data_received_callback(callback)
   Adds a callback for the event MicroPythonDataReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives one argument.
   • The data received as a Bytearray

add_socket_state_received_callback(callback)
   Adds a callback for the event SocketStateReceived.
   
   Parameters callback (Function or List of functions) – the callback. Receives two arguments.
• The socket ID as an Integer.
• The state received as a SocketState

**add_socket_data_received_callback** *(callback)*
Adds a callback for the event *SocketDataReceived.*

**Parameters**

**callback** *(Function or List of functions)* – the callback. Receives two arguments.

• The socket ID as an Integer.
• The status received as a SocketStatus

**add_socket_data_received_from_callback** *(callback)*
Adds a callback for the event *SocketDataReceivedFrom.*

**Parameters**

**callback** *(Function or List of functions)* – the callback. Receives three arguments.

• The socket ID as an Integer.
• A pair (host, port) of the source address where host is a string representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
• The status received as a SocketStatus

**add_route_record_received_callback** *(callback)*
Adds a callback for the event *RouteRecordIndicatorReceived.*

**Parameters**

• **callback** *(Function or List of functions)* – the callback. Receives two arguments.

• **arguments** –
  
  – *Source* *(RemoteXBeeDevice):* The remote node that sent the route record.
  
  – *Hops* *(List):* List of intermediate hops 16-bit addresses from closest to source (who sent the route record) to closest to destination.

**add_route_info_received_callback** *(callback)*
Adds a callback for the event *RouteInformationReceived.*

**Parameters**

• **callback** *(Function or List of functions)* – the callback. Receives eight arguments.

• **arguments** –

  – *Source event* *(Integer):* The source event (0x11: NACK, 0x12: Trace route)
  
  – *Timestamp* *(Integer):* The system timer value on the node generating this package. The timestamp is in microseconds.
  
  – *ACK timeout count* *(Integer):* Number of MAC ACK timeouts that occur.
  
  – *TX blocked count* *(Integer):* Number of times the transmissions was blocked due to reception in progress.
  
  – *Destination address* *(XBee64BitAddress):* 64-bit address of the final destination node.
– Source address (**XBee64BitAddress**): 64-bit address of the source node.

– Responder address (**XBee64BitAddress**): 64-bit address of the node that generates this packet after it sends (or attempts to send) the packet to the next hop (successor node).

– Successor address (**XBee64BitAddress**): 64-bit address of the next node after the responder in the route towards the destination.

**del_packet_received_callback**(callback)
Deletes a callback for the callback list of **PacketReceived** event.

**Parameters**

`callback` (*Function*) – the callback to delete.

**Raises**

`ValueError` – if `callback` is not in the callback list of **PacketReceived** event.

**del_data_received_callback**(callback)
Deletes a callback for the callback list of **DataReceived** event.

**Parameters**

`callback` (*Function*) – the callback to delete.

**Raises**

`ValueError` – if `callback` is not in the callback list of **DataReceived** event.

**del_modem_status_received_callback**(callback)
Deletes a callback for the callback list of **ModemStatusReceived** event.

**Parameters**

`callback` (*Function*) – the callback to delete.

**Raises**

`ValueError` – if `callback` is not in the callback list of **ModemStatusReceived** event.

**del_io_sample_received_callback**(callback)
Deletes a callback for the callback list of **IOSampleReceived** event.

**Parameters**

`callback` (*Function*) – the callback to delete.

**Raises**

`ValueError` – if `callback` is not in the callback list of **IOSampleReceived** event.

**del_explicit_data_received_callback**(callback)
Deletes a callback for the callback list of **ExplicitDataReceived** event.

**Parameters**

`callback` (*Function*) – the callback to delete.

**Raises**

`ValueError` – if `callback` is not in the callback list of **ExplicitDataReceived** event.

**del_ip_data_received_callback**(callback)
Deletes a callback for the callback list of **IPDataReceived** event.

**Parameters**

`callback` (*Function*) – the callback to delete.

**Raises**

`ValueError` – if `callback` is not in the callback list of **IPDataReceived** event.

**del_sms_received_callback**(callback)
Deletes a callback for the callback list of **SMSReceived** event.

**Parameters**

`callback` (*Function*) – the callback to delete.

**Raises**

`ValueError` – if `callback` is not in the callback list of **SMSReceived** event.

**del_user_data_relay_received_callback**(callback)
Deletes a callback for the callback list of **RelayDataReceived** event.
Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of RelayDataReceived event.

**del_bluetooth_data_received_callback** *(callback)*
Deletes a callback for the callback list of BluetoothDataReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of BluetoothDataReceived event.

**del_micropython_data_received_callback** *(callback)*
Deletes a callback for the callback list of MicroPythonDataReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of MicroPythonDataReceived event.

**del_socket_state_received_callback** *(callback)*
Deletes a callback for the callback list of SocketStateReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of SocketStateReceived event.

**del_socket_data_received_callback** *(callback)*
Deletes a callback for the callback list of SocketDataReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of SocketDataReceived event.

**del_socket_data_received_from_callback** *(callback)*
Deletes a callback for the callback list of SocketDataReceivedFrom event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of SocketDataReceivedFrom event.

**del_route_record_received_callback** *(callback)*
Deletes a callback for the callback list of RouteRecordIndicatorReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of RouteRecordIndicatorReceived event.

**del_route_info_callback** *(callback)*
Deletes a callback for the callback list of RouteInformationReceived event.

Parameters callback (Function) – the callback to delete.

Raises ValueError – if callback is not in the callback list of RouteInformationReceived event.

**get_packet_received_callbacks** ()
Returns the list of registered callbacks for received packets.

Returns List of PacketReceived events.
Return type List

get_data_received_callbacks() Returns the list of registered callbacks for received data.

Returns List of DataReceived events.

Return type List

get_modem_status_received_callbacks() Returns the list of registered callbacks for received modem status.

Returns List of ModemStatusReceived events.

Return type List

get_io_sample_received_callbacks() Returns the list of registered callbacks for received IO samples.

Returns List of IOSampleReceived events.

Return type List

get_explicit_data_received_callbacks() Returns the list of registered callbacks for received explicit data.

Returns List of ExplicitDataReceived events.

Return type List

get_ip_data_received_callbacks() Returns the list of registered callbacks for received IP data.

Returns List of IPDataReceived events.

Return type List

get_sms_received_callbacks() Returns the list of registered callbacks for received SMS.

Returns List of SMSReceived events.

Return type List

get_user_data_relay_received_callbacks() Returns the list of registered callbacks for received user data relay.

Returns List of RelayDataReceived events.

Return type List

get_bluetooth_data_received_callbacks() Returns the list of registered callbacks for received Bluetooth data.

Returns List of BluetoothDataReceived events.

Return type List

get_micropython_data_received_callbacks() Returns the list of registered callbacks for received micropython data.

Returns List of MicroPythonDataReceived events.

Return type List

get_socket_state_received_callbacks() Returns the list of registered callbacks for received socket state.
Returns List of `SocketStateReceived` events.

Return type List

def get_socket_data_received_callbacks():
    Returns the list of registered callbacks for received socket data.

    Returns List of `SocketDataReceived` events.

    Return type List

def get_socket_data_received_from_callbacks():
    Returns the list of registered callbacks for received socket data from.

    Returns List of `SocketDataReceivedFrom` events.

    Return type List

def get_route_record_received_callbacks():
    Returns the list of registered callbacks for received route records.

    Returns List of `RouteRecordIndicatorReceived` events.

    Return type List

def get_route_info_callbacks():
    Returns the list of registered callbacks for received route information packets.

    Returns List of `RouteInformationReceived` events.

    Return type List

ident
    Thread identifier of this thread or None if it has not been started.

    This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread
    exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
    Return whether the thread is alive.

    This method is deprecated, use is_alive() instead.

is_alive()
    Return whether the thread is alive.

    This method returns True just before the run() method starts until just after the run() method terminates.
    The module function enumerate() returns a list of all alive threads.

join(timeout=None)
    Wait until the thread terminates.

    This blocks the calling thread until the thread whose join() method is called terminates – either normally
    or through an unhandled exception or until the optional timeout occurs.

    When the timeout argument is present and not None, it should be a floating point number specifying a
    timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call
    is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call
    timed out.

    When the timeout argument is not present or None, the operation will block until the thread terminates.
    A thread can be join()ed many times.
join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

**name**
A string used for identification purposes only.
It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

**start()**
Start the thread’s activity.
It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

```python
class digi.xbee.reader.XBeeQueue(maxsize=10)
    Bases: queue.Queue

    This class represents an XBee queue.

    Class constructor. Instantiates a new XBeeQueue with the provided parameters.

    Parameters (Integer, default (maxsize) – 10) the maximum size of the queue.

    get (block=True, timeout=None)
        Returns the first element of the queue if there is some element ready before timeout expires, in case of the timeout is not None.

        If timeout is None, this method is non-blocking. In this case, if there isn’t any element available, it returns None, otherwise it returns an XBeeAPIPacket.

        Parameters

        • block (Boolean) – True to block during timeout waiting for a packet, False to not block.

        • timeout (Integer, optional) – timeout in seconds.

        Returns

        a packet if there is any packet available before timeout expires. If timeout is None, the returned value may be None.

        Return type XBeeAPIPacket

        Raises TimeoutException – if timeout is not None and there isn’t any packet available before the timeout expires.

    get_by_remote(remote_xbee_device, timeout=None)
        Returns the first element of the queue that had been sent by remote_xbee_device, if there is some in the specified timeout.

        If timeout is None, this method is non-blocking. In this case, if there isn’t any packet sent by remote_xbee_device in the queue, it returns None, otherwise it returns an XBeeAPIPacket.

        Parameters

        • remote_xbee_device (RemoteXBeeDevice) – the remote XBe device to get its firs element from queue.

        • timeout (Integer, optional) – timeout in seconds.
```
Returns

if there is any packet available before the timeout expires. If timeout is None, the returned value may be None.

Return type XBeeAPIPacket

Raises TimeoutException – if timeout is not None and there isn’t any packet available that has been sent by remote_xbee_device before the timeout expires.

get_by_ip(ip_addr, timeout=None)

Returns the first IP data packet from the queue whose IP address matches the provided address.

If timeout is None, this method is non-blocking. In this case, if there isn’t any packet sent by remote_xbee_device in the queue, it returns None, otherwise it returns an XBeeAPIPacket.

Parameters

• ip_addr (ipaddress.IPv4Address) – The IP address to look for in the list of packets.

• timeout (Integer, optional) – Timeout in seconds.

Returns

if there is any packet available before the timeout expires. If timeout is None, the returned value may be None.

Return type XBeeAPIPacket

Raises TimeoutException – if timeout is not None and there isn’t any packet available that has been sent by remote_xbee_device before the timeout expires.

get_by_id(frame_id, timeout=None)

Returns the first packet from the queue whose frame ID matches the provided one.

If timeout is None, this method is non-blocking. In this case, if there isn’t any received packet with the provided frame ID in the queue, it returns None, otherwise it returns an XBeeAPIPacket.

Parameters

• frame_id (Integer) – The frame ID to look for in the list of packets.

• timeout (Integer, optional) – Timeout in seconds.

Returns

if there is any packet available before the timeout expires. If timeout is None, the returned value may be None.

Return type XBeeAPIPacket

Raises

• TimeoutException – if timeout is not None and there isn’t any packet available that matches

• the provided frame ID before the timeout expires.

empty()

Return True if the queue is empty. False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.
To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the `join()` method.

**flush()**
Cancels the task.

**full()**
Return True if the queue is full, False otherwise (not reliable!).
This method is likely to be removed at some point. Use `qsize() >= n` as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of `full()` or `qsize()` can be used.

**get_nowait()**
Remove and return an item from the queue without blocking.
Only get an item if one is immediately available. Otherwise raise the Empty exception.

**join()**
Blocks until all items in the Queue have been gotten and processed.
The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate the item was retrieved and all work on it is complete.
When the count of unfinished tasks drops to zero, `join()` unblocks.

**put(item, block=True, timeout=None)**
Put an item into the queue.
If optional args ‘block’ is true and ‘timeout’ is None (the default), block if necessary until a free slot is available. If ‘timeout’ is a non-negative number, it blocks at most ‘timeout’ seconds and raises the Full exception if no free slot was available within that time. Otherwise (‘block’ is false), put an item on the queue if a free slot is immediately available, else raise the Full exception (‘timeout’ is ignored in that case).

**put_nowait(item)**
Put an item into the queue without blocking.
Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

**qsize()**
Return the approximate size of the queue (not reliable!).

**task_done()**
Indicate that a formerly enqueued task is complete.
Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.
If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been put() into the queue).
Raises a ValueError if called more times than there were items placed in the queue.

---

**digi.xbee.recovery module**

**digi.xbee.recovery.recover_device(target)**
Recovers the XBee from an unknown state and leaves if configured for normal operations.

Parameters:
**target** (String or XBeeDevice) – target of the recovery operation.

Raises **RecoveryException** – if there is any error performing the recovery action.
**digi.xbee.serial module**

```python
class digi.xbee.serial.FlowControl
    Bases: enum.Enum
    This class represents all available flow controls.

class digi.xbee.serial.XBeeSerialPort
    Bases: sphinx.ext.autodoc.importer._MockObject, digi.xbee.comm_interface.XBeeCommunicationInterface
    This class extends the functionality of Serial class (PySerial).
```

It also introduces a minor change in its behaviour: the serial port is not automatically open when an object is instantiated, only when calling open().

**See also:**

`_PySerial`: https://github.com/pyserial/pyserial

Class constructor. Instantiates a new XBeeSerialPort object with the given port parameters.

**Parameters**

- **baud_rate** (Integer) – serial port baud rate.
- **port** (String) – serial port name to use.
- **data_bits** (Integer, optional) – serial data bits. Default to 8.
- **stop_bits** (Float, optional) – serial stop bits. Default to 1.
- **parity** (Char, optional) – serial parity. Default to ‘N’ (None).
- **flow_control** (Integer, optional) – serial flow control. Default to None.
- **timeout** (Integer, optional) – read timeout. Default to 0.1 seconds.

**See also:**

`_PySerial`: https://github.com/pyserial/pyserial

**open()**

Opens port with current settings. This may throw a SerialException if the port cannot be opened.

**is_interface_open**

Returns whether the underlying hardware communication interface is active or not.

**Returns** Boolean. True if the interface is active, False otherwise.

**write_frame** (frame)

 Writes an XBee frame to the underlying hardware interface.

Subclasses may throw specific exceptions to signal implementation specific hardware errors.
**Parameters** frame (Bytearray) – The XBee API frame packet to write. If the bytearray does not correctly represent an XBee frame, the behaviour is undefined.

**read_byte()**
Synchronous. Reads one byte from serial port.
- **Returns** the read byte.
- **Return type** Integer
- **Raises** `TimeoutException` – if there is no bytes ins serial port buffer.

**read_bytes**(num_bytes)
Synchronous. Reads the specified number of bytes from the serial port.
- **Parameters** num_bytes (Integer) – the number of bytes to read.
- **Returns** the read bytes.
- **Return type** Bytearray
- **Raises** `TimeoutException` – if the number of bytes read is less than num_bytes.

**quit_reading()**
Makes the thread (if any) blocking on wait_for_frame return.
If a thread was blocked on wait_for_frame, this method blocks (for a maximum of ‘timeout’ seconds) until the blocked thread is resumed.

**wait_for_frame**(operating_mode=<OperatingMode.API_MODE: (1, 'API mode')>)
Reads the next packet. Starts to read when finds the start delimiter. The last byte read is the checksum.
If there is something in the COM buffer after the start delimiter, this method discards it.
If the method can’t read a complete and correct packet, it will return None.
- **Parameters** operating_mode (OperatingMode) – the operating mode in which the packet should be read.
- **Returns** the read packet as bytearray if a packet is read, None otherwise.
- **Return type** Bytearray

**read_existing()**
Asynchronous. Reads all bytes in the serial port buffer. May read 0 bytes.
- **Returns** the bytes read.
- **Return type** Bytearray

**get_read_timeout()**
Returns the serial port read timeout.
- **Returns** read timeout in seconds.
- **Return type** Integer

**set_read_timeout**(read_timeout)
Sets the serial port read timeout in seconds.
- **Parameters** read_timeout (Integer) – the new serial port read timeout in seconds.

**set_baudrate**(new_baudrate)
Changes the serial port baudrate.
- **Parameters** new_baudrate (Integer) – the new baudrate to set.
purge_port()
    Purges the serial port by cleaning the input and output buffers.

close()
    Terminates the underlying hardware communication interface.
    Subclasses may throw specific exceptions to signal implementation specific hardware errors.

timeout
    Returns the read timeout.
    Returns  read timeout in seconds.
    Return type  Integer

digi.xbee.xsocket module

class digi.xbee.xsocket.socket (xbee_device, ip_protocol=<IPProtocol.TCP: (1, 'TCP')>)
    Bases: object
    This class represents an XBee socket and provides methods to create, connect, bind and close a socket, as well as send and receive data with it.
    Class constructor. Instantiates a new XBee socket object for the given XBee device.
    Parameters
    • xbee_device (XBeeDevice) – XBee device of the socket.
    • ip_protocol (IPProtocol) – protocol of the socket.
    Raises
    • ValueError – if xbee_device is None or if xbee_device is not an instance of CellularDevice.
    • ValueError – if ip_protocol is None.
    • XBeeException – if the connection with the XBee device is not open.
    connect (address)
    Connects to a remote socket at the given address.
    Parameters address (Tuple) – A pair (host, port) where host is the domain name or string representation of an IPv4 and port is the numeric port value.
    Raises
    • TimeoutException – if the connect response is not received in the configured timeout.
    • ValueError – if address is None or not a pair (host, port).
    • ValueError – if port is less than 1 or greater than 65535.
    • XBeeException – if the connection with the XBee device is not open.
    • XBeeSocketException – if the connect status is not SUCCESS.

bind (address)
    Binds the socket to the given address. The socket must not already be bound.
    Parameters address (Tuple) – A pair (host, port) where host is the local interface (not used) and port is the numeric port value.
    Raises
• **TimeoutException** – if the bind response is not received in the configured timeout.

• **ValueError** – if address is None or not a pair (host, port).

• **ValueError** – if port is less than 1 or greater than 65535.

• **XBeeException** – if the connection with the XBee device is not open.

• **XBeeSocketException** – if the bind status is not **SUCCESS**.

• **XBeeSocketException** – if the socket is already bound.

**listen**(backlog=1)

Enables a server to accept connections.

Parameters **backlog** (Integer, optional) – The number of unaccepted connections that the system will allow before refusing new connections. If specified, it must be at least 0 (if it is lower, it is set to 0).

Raises **XBeeSocketException** – if the socket is not bound.

**accept**( )

Accepts a connection. The socket must be bound to an address and listening for connections.

Returns

A pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is a pair (host, port) with the address bound to the socket on the other end of the connection.

Return type Tuple

Raises

• **XBeeException** – if the connection with the XBee device is not open.

• **XBeeSocketException** – if the socket is not bound or not listening.

**gettimeout**( )

Returns the configured socket timeout in seconds.

Returns the configured timeout in seconds.

Return type Integer

**settimeout**(timeout)

Sets the socket timeout in seconds.

Parameters **timeout** (Integer) – the new socket timeout in seconds.

**getblocking**( )

Returns whether the socket is in blocking mode or not.

Returns True if the socket is in blocking mode, False otherwise.

Return type Boolean

**setblocking**(flag)

Sets the socket in blocking or non-blocking mode.

Parameters **flag** (Boolean) – True to set the socket in blocking mode, False to set it in no block mode and configure the timeout with the default value (5 seconds).

**recv**(bufsize)

Receives data from the socket.
Parameters **bufsize** (*Integer*) – The maximum amount of data to be received at once.

Returns the data received.

Return type *Bytearray*

Raises **ValueError** – if `bufsize` is less than 1.

recvfrom (**bufsize**) Receives data from the socket.

Parameters **bufsize** (*Integer*) – the maximum amount of data to be received at once.

Returns

Pair containing the data received (*Bytearray*) and the address of the socket sending the data. The address is also a pair (*host*, *port*) where *host* is the string representation of an IPv4 and *port* is the numeric port value.

Return type *Tuple (Bytearray, Tuple)*

Raises **ValueError** – if `bufsize` is less than 1.

send (**data**) Sends data to the socket and returns the number of bytes sent. The socket must be connected to a remote socket. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

Parameters **data** (*Bytearray*) – the data to send.

Returns the number of bytes sent.

Return type *Integer*

Raises

- **ValueError** – if the data to send is `None`.
- **ValueError** – if the number of bytes to send is 0.
- **XBeeException** – if the connection with the XBee device is not open.
- **XBeeSocketException** – if the socket is not valid.
- **XBeeSocketException** – if the socket is not open.

sendall (**data**) Sends data to the socket. The socket must be connected to a remote socket. Unlike `send()`, this method continues to send data from bytes until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Parameters **data** (*Bytearray*) – the data to send.

Raises

- **TimeoutException** – if the send status response is not received in the configured timeout.
- **ValueError** – if the data to send is `None`.
- **ValueError** – if the number of bytes to send is 0.
- **XBeeException** – if the connection with the XBee device is not open.
- **XBeeSocketException** – if the socket is not valid.
- **XBeeSocketException** – if the send status is not SUCCESS.
**sendto** *(data, address)*

Sends data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*.

**Parameters**

- **data** *(Bytearray)* – the data to send.
- **address** *(Tuple)* – the address of the destination socket. It must be a pair *(host, port)* where host is the domain name or string representation of an IPv4 and port is the numeric port value.

**Returns** the number of bytes sent.

**Return type** Integer

**Raises**

- **TimeoutException** – if the send status response is not received in the configured timeout.
- **ValueError** – if the data to send is *None*.
- **ValueError** – if the number of bytes to send is 0.
- **XBeeException** – if the connection with the XBee device is not open.
- **XBeeSocketException** – if the socket is already open.
- **XBeeSocketException** – if the send status is not **SUCCESS**.

**close()**

Closes the socket.

**Raises**

- **TimeoutException** – if the close response is not received in the configured timeout.
- **XBeeException** – if the connection with the XBee device is not open.
- **XBeeSocketException** – if the close status is not **SUCCESS**.

**setsockopt** *(option, value)*

Sets the value of the given socket option.

**Parameters**

- **option** *(SocketOption)* – the socket option to set its value.
- **value** *(Bytearray)* – the new value of the socket option.

**Raises**

- **TimeoutException** – if the socket option response is not received in the configured timeout.
- **ValueError** – if the option to set is *None*.
- **ValueError** – if the value of the option is *None*.
- **XBeeException** – if the connection with the XBee device is not open.
- **XBeeSocketException** – if the socket option response status is not **SUCCESS**.
**getsockopt** *(option)*

Returns the value of the given socket option.

**Parameters**

*option*(SocketOption) – the socket option to get its value.

**Returns**

the value of the socket option.

**Return type**

Bytearray

**Raises**

- *TimeoutException* – if the socket option response is not received in the configured timeout.
- *ValueError* – if the option to set is None.
- *XBeeException* – if the connection with the XBee device is not open.
- *XBeeSocketException* – if the socket option response status is not SUCCESS.

**add_socket_state_callback** *(callback)*

Adds a callback for the event *digi.xbee.reader.SocketStateReceived*.

**Parameters**

*callback*(Function) – the callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a SocketState

**del_socket_state_callback** *(callback)*

Deletes a callback for the callback list of *digi.xbee.reader.SocketStateReceived* event.

**Parameters**

*callback*(Function) – the callback to delete.

**Raises**

*ValueError* – if *callback* is not in the callback list of *digi.xbee.reader.SocketStateReceived* event.

**get_sock_info**

Returns the information of this socket.

**Returns**

The socket information.

**Return type**

SocketInfo

**Raises**

- *InvalidOperatingModeException* – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.

**See also:**

SocketInfo

**is_connected**

Boolean. Indicates whether the socket is connected or not.
CHAPTER 3

Indices and tables

• genindex
• modindex
• search
Copyright 2017-2019, Digi International Inc.

This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, you can obtain one at http://mozilla.org/MPL/2.0/.

THE SOFTWARE IS PROVIDED “AS IS” AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Digi International Inc. 11001 Bren Road East, Minnetonka, MN 55343
d

digi, 114

digi.xbee, 114

digi.xbee.comm_interface, 322

digi.xbee.devices, 323

digi.xbee.exception, 723

digi.xbee.filesystem, 725

digi.xbee.firmware, 729

digi.xbee.io, 730

digi.xbee.models, 114

digi.xbee.models.accesspoint, 114

digi.xbee.models.address, 123

digi.xbee.models.atcomm, 115

digi.xbee.models.hw, 119

digi.xbee.models.message, 126

digi.xbee.models.mode, 121

digi.xbee.models.options, 130

digi.xbee.models.protocol, 135

digi.xbee.models.status, 137

digi.xbee.packets, 147

digi.xbee.packets.aft, 147

digi.xbee.packets.base, 149

digi.xbee.packets.cellular, 157

digi.xbee.packets.common, 163

digi.xbee.packets.devicecloud, 201

digi.xbee.packets.digimesh, 217

digi.xbee.packets.factory, 318

digi.xbee.packets.network, 222

digi.xbee.packets.raw, 228

digi.xbee.packets.relay, 248

digi.xbee.packets.socket, 254

digi.xbee.packets.wifi, 299

digi.xbee.packets.zigbee, 308

digi.xbee.profile, 735

digi.xbee.reader, 742

digi.xbee.recovery, 771

digi.xbee.serial, 772

digi.xbee.util, 318

digi.xbee.util.utils, 318

digi.xbee.xsocket, 774
AbstractXBeeDevice (class in digi.xbee.devices), 323
accept() (digi.xbee.xsocket.socket method), 775
AccessPoint (class in digi.xbee.models.accesspoint), 114
ack_timeout_count
(digi.xbee.packets.digimesh.RouteInformationPacket attribute), 219
ADC (digi.xbee.io.IOMode attribute), 735
add_bluetooth_data_received_callback()
(digi.xbee.devices.CellularDevice method), 496
add_bluetooth_data_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 389
add_bluetooth_data_received_callback()
(digi.xbee.devices.DigiPointDevice method), 414
add_bluetooth_data_received_callback()
(digi.xbee.devices.IPDevice method), 473
add_bluetooth_data_received_callback()
(digi.xbee.devices.LPWANDevice method), 518
add_bluetooth_data_received_callback()
(digi.xbee.devices.NBIoTDevice method), 542
add_bluetooth_data_received_callback()
(digi.xbee.devices.Raw802Device method), 364
add_bluetooth_data_received_callback()
(digi.xbee.devices.WiFiDevice method), 571
add_bluetooth_data_received_callback()
(digi.xbee.devices.XBeeDevice method), 344
add_bluetooth_data_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 445
add_device_discovered_callback()
(digi.xbee.devices.DigiMeshNetwork method), 700
add_device_discovered_callback()
(digi.xbee.devices.DigiPointNetwork method), 710
add_device_discovered_callback()
(digi.xbee.devices.Raw802Network method), 691
add_device_discovered_callback()
(digi.xbee.devices.XBeeNetwork method), 674
add_device_discovered_callback()
(digi.xbee.devices.ZigBeeNetwork method), 762
add_data_received_callback()
(digi.xbee.devices.DigiPointDevice method), 414
add_data_received_callback()
(digi.xbee.devices.IPDevice method), 472
add_data_received_callback()
(digi.xbee.devices.LPWANDevice method), 518
add_data_received_callback()
(digi.xbee.devices.NBIoTDevice method), 542
add_data_received_callback()
(digi.xbee.devices.Raw802Device method), 364
add_data_received_callback()
(digi.xbee.devices.WiFiDevice method), 571
add_data_received_callback()
(digi.xbee.devices.XBeeDevice method), 344
add_data_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 445
add_data_received_callback()
(digi.xbee.reader.PacketListener method), 762
add_data_received_callback()
(digi.xbee.devices.DigiMeshNetwork method), 700
add_data_received_callback()
(digi.xbee.devices.DigiPointNetwork method), 710
add_data_received_callback()
(digi.xbee.devices.Raw802Network method), 691
add_data_received_callback()
(digi.xbee.devices.XBeeNetwork method), 674
add_data_received_callback()
(digi.xbee.devices.ZigBeeNetwork method), 785
add_discovery_process_finished_callback()
(digi.xbee.devices.DigiMeshNetwork method), 681

add_discovery_process_finished_callback()
(digi.xbee.devices.DigiPointNetwork method), 700

add_discovery_process_finished_callback()
(digi.xbee.devices.Raw802Network method), 691

add_discovery_process_finished_callback()
(digi.xbee.devices.XBeeNetwork method), 675

add_discovery_process_finished_callback()
(digi.xbee.devices.ZigBeeNetwork method), 682

add_end_discovery_scan_callback()
(digi.xbee.devices.DigiMeshNetwork method), 681

add_end_discovery_scan_callback()
(digi.xbee.devices.DigiPointNetwork method), 701

add_end_discovery_scan_callback()
(digi.xbee.devices.Raw802Network method), 710

add_end_discovery_scan_callback()
(digi.xbee.devices.XBeeNetwork method), 691

add_end_discovery_scan_callback()
(digi.xbee.devices.ZigBeeNetwork method), 710

add_explicit_data_received_callback()
(digi.xbee.reader.PacketListener method), 763

add_explicit_data_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 496

add_explicit_data_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 390

add_explicit_data_received_callback()
(digi.xbee.devices.DigiPointDevice method), 415

add_explicit_data_received_callback()
(digi.xbee.devices.IPDevice method), 472

add_explicit_data_received_callback()
(digi.xbee.devices.LPWANDevice method), 518

add_explicit_data_received_callback()
(digi.xbee.devices.NBIoTDevice method), 542

add_explicit_data_received_callback()
(digi.xbee.devices.Raw802Device method), 364

add_explicit_data_received_callback()
(digi.xbee.devices.WiFiDevice method), 572

add_explicit_data_received_callback()
(digi.xbee.devices.XBeeDevice method), 542

add_init_discovery_scan_callback()
(digi.xbee.devices.DigiMeshNetwork method), 701

add_init_discovery_scan_callback()
(digi.xbee.devices.DigiPointNetwork method), 711

add_init_discovery_scan_callback()
(digi.xbee.devices.Raw802Network method), 692

add_init_discovery_scan_callback()
(digi.xbee.devices.XBeeNetwork method), 674

add_init_discovery_scan_callback()
(digi.xbee.devices.ZigBeeNetwork method), 682

add_io_sample_received_callback()
(digi.xbee.devices.CellularDevice method), 495

add_io_sample_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 390

add_io_sample_received_callback()
(digi.xbee.devices.DigiPointDevice method), 415

add_io_sample_received_callback()
(digi.xbee.devices.IPDevice method), 473

add_io_sample_received_callback()
(digi.xbee.devices.LPWANDevice method), 518

add_io_sample_received_callback()
(digi.xbee.devices.NBIoTDevice method), 542

add_io_sample_received_callback()
(digi.xbee.devices.Raw802Device method),
add_io_sample_received_callback()
(digi.xbee.devices.WiFiDevice method), 572
add_io_sample_received_callback()
(digi.xbee.devices.XBeeDevice method),
344
add_io_sample_received_callback()
(digi.xbee.devices.ZigBeeDevice method),
445
add_io_sample_received_callback()
(digi.xbee.reader.PacketListener method),
762
add_ip_data_received_callback()
(digi.xbee.devices.CellularDevice method),
496
add_ip_data_received_callback()
(digi.xbee.devices.IPDevice method), 469
add_ip_data_received_callback()
(digi.xbee.devices.LPWANDevice method),
518
add_ip_data_received_callback()
(digi.xbee.devices.NBIoTDevice method),
542
add_ip_data_received_callback()
(digi.xbee.devices.WiFiDevice method), 572
add_ip_data_received_callback()
(digi.xbee.devices.XBeeDevice method), 344
add_ip_data_received_callback()
(digi.xbee.devices.ZigBeeDevice method),
445
add_micropython_data_received_callback()
(digi.xbee.devices.CellularDevice method),
390
add_micropython_data_received_callback()
(digi.xbee.devices.DigiMeshDevice method),
390
add_micropython_data_received_callback()
(digi.xbee.devices.DigiPointDevice method),
415
add_micropython_data_received_callback()
(digi.xbee.devices.IPDevice method), 473
add_micropython_data_received_callback()
(digi.xbee.devices.LPWANDevice method),
415
add_micropython_data_received_callback()
(digi.xbee.devices.NBIoTDevice method),
445
add_micropython_data_received_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_micropython_data_received_callback()
(digi.xbee.devices.ZigBeeDevice method),
445
add_micropython_data_received_callback()
(digi.xbee.reader.PacketListener method),
763
add_modem_status_received_callback()
(digi.xbee.devices.CellularDevice method), 496
add_modem_status_received_callback()
(digi.xbee.devices.DigiMeshDevice method),
390
add_modem_status_received_callback()
(digi.xbee.devices.DigiPointDevice method),
415
add_modem_status_received_callback()
(digi.xbee.devices.IPDevice method), 473
add_modem_status_received_callback()
(digi.xbee.devices.LPWANDevice method),
519
add_modem_status_received_callback()
(digi.xbee.devices.NBIoTDevice method),
542
add_modem_status_received_callback()
(digi.xbee.devices.Raw802Device method), 364
add_modem_status_received_callback()
(digi.xbee.devices.WiFiDevice method), 572
add_modem_status_received_callback()
(digi.xbee.devices.XBeeDevice method), 344
add_modem_status_received_callback()
(digi.xbee.devices.ZigBeeDevice method),
446
add_network_modified_callback()
(digi.xbee.devices.DigiMeshNetwork method),
702
add_network_modified_callback()
(digi.xbee.devices.DigiPointNetwork method),
711
add_network_modified_callback()
(digi.xbee.devices.Raw802Network method),
692
add_network_modified_callback()
(digi.xbee.devices.XBeeNetwork method),
673
add_network_modified_callback()
(digi.xbee.devices.ZigBeeNetwork method),
682
add_packet_received_callback()
(digi.xbee.devices.CellularDevice method),
496
add_packet_received_callback()
(digi.xbee.devices.DigiMeshDevice method),
390
add_packet_received_callback()
(digi.xbee.devices.DigiPointDevice method),
415
add_packet_received_callback()
(digi.xbee.devices.IPDevice method), 473
add_packet_received_callback()
(digi.xbee.devices.LPWANDevice method), 518
add_packet_received_callback()
(digi.xbee.devices.NBIoTDevice method),
542
add_packet_received_callback()
(digi.xbee.devices.Raw802Device method), 364
add_packet_received_callback()
(digi.xbee.devices.WiFiDevice method), 572
add_packet_received_callback()
(digi.xbee.devices.XBeeDevice method), 344
add_packet_received_callback()
(digi.xbee.devices.ZigBeeDevice method),
445
add_packet_received_callback()
(digi.xbee.reader.PacketListener method), 763
add_modem_status_received_callback()
(digi.xbee.devices.CellularDevice method), 496
add_packet_received_callback()
  (digi.xbee.devices.Raw802Device method), 364
add_packet_received_callback()
  (digi.xbee.devices.WiFiDevice method), 572
add_packet_received_callback()
  (digi.xbee.devices.XBeeDevice method), 344
add_packet_received_callback()
  (digi.xbee.devices.ZigBeeDevice method), 446
add_packet_received_callback()
  (digi.xbee.reader.PacketListener method), 762
add_route_received_callback()
  (digi.xbee.devices.WiFiDevice method), 572
add_route_received_callback()
  (digi.xbee.devices.XBeeDevice method), 348
add_route_received_callback()
  (digi.xbee.devices.ZigBeeDevice method), 446
add_route_record_received_callback()
  (digi.xbee.reader.PacketListener method), 764
add_sms_callback()
  (digi.xbee.devices.CellularDevice method), 493
add_sms_callback()
  (digi.xbee.devices.LPWANDevice method), 518
add_sms_callback()
  (digi.xbee.devices.NBIoTDevice method), 543
add_socket_data_received_callback()
  (digi.xbee.devices.CellularDevice method), 497
add_socket_data_received_callback()
  (digi.xbee.devices.DigiMeshDevice method), 390
add_socket_data_received_callback()
  (digi.xbee.devices.DigiPointDevice method), 415
add_socket_data_received_callback()
  (digi.xbee.devices.IPDevice method), 474
add_socket_data_received_callback()
  (digi.xbee.devices.LPWANDevice method), 519
add_socket_data_received_callback()
  (digi.xbee.devices.NBIoTDevice method), 543
add_socket_data_received_callback()
  (digi.xbee.devices.Raw802Device method), 365
add_socket_data_received_callback()
  (digi.xbee.devices.WiFiDevice method), 572
add_socket_data_received_callback()
  (digi.xbee.devices.XBeeDevice method), 345
add_socket_data_received_callback()
  (digi.xbee.devices.ZigBeeDevice method), 446
add_socket_data_received_from_callback()
  (digi.xbee.devices.CellularDevice method), 764
add_socket_data_received_from_callback()
  (digi.xbee.devices.DigiMeshDevice method), 497
add_socket_data_received_from_callback()
  (digi.xbee.devices.DigiPointDevice method), 391
add_socket_data_received_from_callback()
add_user_data_relay_received_callback()
(digi.xbee.devices.DigiPointDevice method), 416
add_socket_data_received_from_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_data_received_from_callback()
(digi.xbee.devices.XBeeDevice method), 497
add_socket_state_callback()
(digi.xbee.xsocket.socket method), 778
add_socket_state_received_callback()
(digi.xbee.devices.CellularDevice method), 497
add_socket_state_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 391
add_socket_state_received_callback()
(digi.xbee.devices.DigiPointDevice method), 416
add_socket_state_received_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_state_received_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_state_received_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_state_received_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_state_received_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_state_received_callback()
(digi.xbee.devices.WiFiDevice method), 573
add_socket_state_received_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_data_received_from_callback() address (digi.xbee.models.address.XBee16BitAddress attribute), 125
add_socket_data_received_from_callback() address (digi.xbee.models.address.XBee64BitAddress attribute), 126
analog_mask (digi.xbee.io.IOSample attribute), 734
analog_values (digi.xbee.io.IOSample attribute), 734
ApiFrameType (class in digi.xbee.packets.aft), 147
APIOutputMode (class in digi.xbee.models.mode), 121
APIOutputModeBit (class in digi.xbee.models.mode), 122
append() (digi.xbee.reader.BluetoothDataReceived method), 752
append() (digi.xbee.reader.DataReceived method), 744
append() (digi.xbee.reader.DeviceDiscovered method), 747
append() (digi.xbee.reader.DiscoveryProcessFinished method), 748
append() (digi.xbee.reader.EndDiscoveryScan method), 760
append() (digi.xbee.reader.ExplicitDataReceived method), 749
append() (digi.xbee.reader.InitDiscoveryScan method), 759
append() (digi.xbee.reader.IOSampleReceived method), 745
append() (digi.xbee.reader.IPDataReceived method), 750
append() (digi.xbee.reader.MicroPythonDataReceived method), 753
append() (digi.xbee.reader.ModemStatusReceived method), 744
append() (digi.xbee.reader.NetworkModified method), 746
append() (digi.xbee.reader.PacketReceived method), 743
append() (digi.xbee.reader.RelayDataReceived method), 751

(digi.xbee.devices.DigiPointDevice method), 416
add_user_data_relay_received_callback()
(digi.xbee.devices.LPWANDevice method), 520
add_user_data_relay_received_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_user_data_relay_received_callback()
(digi.xbee.devices.Raw802Device method), 365
add_user_data_relay_received_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_user_data_relay_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 446
add_socket_state_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 416
add_socket_state_received_callback()
(digi.xbee.devices.DigiPointDevice method), 416
add_socket_state_received_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_state_received_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_state_received_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_state_received_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_state_received_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_state_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 446
add_socket_data_received_from_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_data_received_from_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_data_received_from_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_data_received_from_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_data_received_from_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_data_received_from_callback()
(digi.xbee.devices.ZigBeeDevice method), 446
add_socket_state_callback()
(digi.xbee.xsocket.socket method), 778
add_socket_state_received_callback()
(digi.xbee.devices.CellularDevice method), 497
add_socket_state_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 391
add_socket_state_received_callback()
(digi.xbee.devices.DigiPointDevice method), 416
add_socket_state_received_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_state_received_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_state_received_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_state_received_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_state_received_callback()
(digi.xbee.devices.WiFiDevice method), 573
add_socket_state_received_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_state_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 446
add_socket_data_received_from_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_data_received_from_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_data_received_from_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_data_received_from_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_data_received_from_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_data_received_from_callback()
(digi.xbee.devices.ZigBeeDevice method), 446
add_socket_state_callback()
(digi.xbee.xsocket.socket method), 778
add_socket_state_received_callback()
(digi.xbee.devices.CellularDevice method), 497
add_socket_state_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 391
add_socket_state_received_callback()
(digi.xbee.devices.DigiPointDevice method), 416
add_socket_state_received_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_state_received_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_state_received_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_state_received_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_state_received_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_state_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 446
add_socket_data_received_from_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_data_received_from_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_data_received_from_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_data_received_from_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_data_received_from_callback()
(digi.xbee.devices.XBeeDevice method), 345
add_socket_data_received_from_callback()
(digi.xbee.devices.ZigBeeDevice method), 446
add_socket_state_callback()
(digi.xbee.xsocket.socket method), 778
add_socket_state_received_callback()
(digi.xbee.devices.CellularDevice method), 497
add_socket_state_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 391
add_socket_state_received_callback()
(digi.xbee.devices.DigiPointDevice method), 416
add_socket_state_received_callback()
(digi.xbee.devices.IPDevice method), 474
add_socket_state_received_callback()
(digi.xbee.devices.LPWANDevice method), 519
add_socket_state_received_callback()
(digi.xbee.devices.NBLoTDevice method), 543
add_socket_state_received_callback()
(digi.xbee.devices.Raw802Device method), 365
add_socket_state_received_callback()
append() (digi.xbee.reader.RouteInformationReceived method), 758
append() (digi.xbee.reader.RouteReceived method), 759
append() (digi.xbee.reader.RouteRecordIndicatorReceived method), 757
append() (digi.xbee.reader.SMSPeriodicReceived method), 751
append() (digi.xbee.reader.SocketDataReceived method), 755
append() (digi.xbee.reader.SocketDataReceivedFrom method), 756
append() (digi.xbee.reader.SocketStateReceived method), 754
apply_profile() (digi.xbee.devices.CellularDevice method), 497
apply_profile() (digi.xbee.devices.DigiMeshDevice method), 391
apply_profile() (digi.xbee.devices.DigiPointDevice method), 416
apply_profile() (digi.xbee.devices.NBIoTDevice method), 543
apply_profile() (digi.xbee.devices.Raw802Device method), 365
apply_profile() (digi.xbee.devices.RemoteDigiMeshDevice method), 626
apply_profile() (digi.xbee.devices.RemoteDigiPointDevice method), 641
apply_profile() (digi.xbee.devices.RemoteRaw802Device method), 610
apply_profile() (digi.xbee.devices.ZigBeeDevice method), 596
apply_profile() (digi.xbee.devices.ZigBeeDevice.method), 447
apply_xbee_profile() (in module digi.xbee.profile), 741
APS_ENCRYPTED (digi.xbee.models.options.ReceiveOptions attribute), 130
AsciiToInt (in module digi.xbee.util.utils), 320
AssociationIndicationStatus (class in digi.xbee.models.status), 141
at_command (digi.xbee.io.IOLine attribute), 731
apply_changes() (digi.xbee.devices.AbstractXBeeDevice method), 325
apply_changes() (digi.xbee.devices.CellularDevice method), 497
apply_changes() (digi.xbee.devices.DigiMeshDevice method), 539
apply_changes() (digi.xbee.devices.LPWANDevice method), 520
apply_changes() (digi.xbee.devices.NBIoTDevice method), 543
apply_changes() (digi.xbee.devices.Raw802Device method), 365
apply_changes() (digi.xbee.devices.XBeeDevice method), 573
apply_changes() (digi.xbee.devices.ZigBeeDevice method), 474
apply_changes() (digi.xbee.devices.ZigBeeDevice method), 474
apply_changes() (digi.xbee.devices.ZigBeeDevice.method), 441
apply_changes() (digi.xbee.devices.ZigBeeDevice method), 474
apply_changes() (digi.xbee.devices.ZigBeeDevice method), 474
apply_changes() (digi.xbee.devices.ZigBeeDevice method), 474
apply_changes() (digi.xbee.devices.ZigBeeDevice method), 474
apply_changes() (digi.xbee.devices.XBeeDevice method), 349
apply_changes() (digi.xbee.devices.XBeeDevice method), 447
apply_changes() (digi.xbee.devices.XBeeDevice method), 539
apply_changes() (digi.xbee.devices.XBeeDevice method), 573
apply_changes() (digi.xbee.devices.XBeeDevice method), 573
apply_changes() (digi.xbee.devices.XBeeDevice method), 573
baudrate (digi.xbee.profile.FirmwareBaudrate attribute), 735
bind() (digi.xbee.socket.socket method), 774
BluetoothDataReceived (class in digi.xbee.reader), 752
BROADCAST_ADDRESS (digi.xbee.models.address.XBee16BitAddress attribute), 124
BROADCAST_ADDRESS (digi.xbee.models.address.XBee64BitAddress attribute), 125
BROADCAST_PACKET (digi.xbee.models.options.ReceiveOptions attribute), 130
broadcast_radius (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 195
broadcast_radius (digi.xbee.packets.common.TransmitPacket attribute), 183
build_aggregate_routes() (digi.xbee.devices.DigiMeshDevice method), 387
build_frame() (in module digi.xbee.packets.factory), 318
bytearray_value (digi.xbee.profile.XBeeProfileSetting attribute), 739
bytes_to_int() (in module digi.xbee.util.utils), 320

C
CASCADEx (digi.xbee.models.mode.NeighborDiscoveryMode attribute), 123
CellularAssociationIndicationStatus (class in digi.xbee.models.status), 142
CellularDevice (class in digi.xbee.devices), 492
change_directory() (digi.xbee.filesystem.LocalXBeefFileSystemManager method), 727
channel (digi.xbee.models.accesspoint.AccessPoint attribute), 115
clear() (digi.xbee.devices.DigiMeshNetwork method), 702
clear() (digi.xbee.devices.DigiPointNetwork method), 712
clear() (digi.xbee.devices.Raw802Network method), 693
clear() (digi.xbee.devices.XBeeNetwork method), 676
clear() (digi.xbee.devices.ZigBeeNetwork method), 683
clear() (digi.xbee.reader.BlueoothDataReceived method), 752
clear() (digi.xbee.reader.DataReceived method), 744
clear() (digi.xbee.reader.DeviceDiscovered method), 747
clear() (digi.xbee.reader.DiscoveryProcessFinished method), 748
clear() (digi.xbee.reader.EndDiscoveryScan method), 760
clear() (digi.xbee.reader.ExplicitDataReceived method), 749
clear() (digi.xbee.reader.InitDiscoveryScan method), 759
clear() (digi.xbee.reader.IOSampleReceived method), 745
clear() (digi.xbee.reader.IPDataReceived method), 750
clear() (digi.xbee.reader.MicroPythonDataReceived method), 753
clear() (digi.xbee.reader.ModemStatusReceived method), 744
clear() (digi.xbee.reader.NetworkModified method), 746
change() (digi.xbee.reader.PacketReceived method), 743
clear() (digi.xbee.reader.RelayDataReceived method), 752
clear() (digi.xbee.reader.RouteInformationReceived method), 758
clear() (digi.xbee.reader.RouteReceived method), 759
clear() (digi.xbee.reader.RouteRecordIndicatorReceived method), 757
clear() (digi.xbee.reader.SMSSent method), 751
clear() (digi.xbee.reader.SocketDataReceived method), 755
clear() (digi.xbee.reader.SocketDataReceivedFrom method), 756
clear() (digi.xbee.reader.SocketStateReceived method), 754
clear() (digi.xbee.reader.XBeeEvent method), 742
clear() (digi.xbee.reader.XBeeEvent method), 742
client_socket_id (digi.xbee.packets.socket.SocketNewIPv4ClientPacket attribute), 289
close() (digi.xbee.comm_interface.XBeeCommunicationInterface method), 322
close() (digi.xbee.devices.CellularDevice method), 498
close() (digi.xbee.devices.DigiMeshDevice method), 392
close() (digi.xbee.devices.DigiMeshDevice method), 392
close() (digi.xbee.devices.DigiPointDevice method), 417
close() (digi.xbee.devices.IPDevice method), 475
close() (digi.xbee.devices.LP WANDevice method), 520
close() (digi.xbee.devices.NBIoTDevice method), 544
close() (digi.xbee.devices.Raw802Device method), 366
close() (digi.xbee.devices.WiFiDevice method), 574
close() (digi.xbee.devices.XBeeDevice method), 340
close() (digi.xbee.devices.ZigBeeDevice method), 447
close() (digi.xbee.serial.XBeeSerialPort method), 774
close() (digi.xbee.xsocket.socket method), 777
cluster_id (digi.xbee.models.message.ExplicitXBeeMessage attribute), 127
code (digi.xbee.models.status.NetworkDiscoveryStatus attribute), 140
code (digi.xbee.models.status.PowerLevel attribute), 141
code (digi.xbee.models.status.SocketInfoState attribute), 144

cluster_id (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 197
code (digi.xbee.models.options.SendDataRequestOptions attribute), 145
code (digi.xbee.models.status.WiFiAssociationIndicationStatus attribute), 147

code (digi.xbee.packets.common.ExplicitRXIndicatorPacket attribute), 201
code (digi.xbee.devices.NetworkEventReason attribute), 146
code (digi.xbee.models.status.SocketState attribute), 146

code (digi.xbee.io.IOValue attribute), 731
code (digi.xbee.models.accesspoint.WiFiEncryptionType attribute), 739

code (digi.xbee.models.atcomm.SpecialByte attribute), 115
code (digi.xbee.models.status.ModemStatus attribute), 118

code (digi.xbee.models.hw.HardwareVersion attribute), 120
code (digi.xbee.models.mode.APIOutputMode attribute), 121
code (digi.xbee.models.mode.APIOutputModeBit attribute), 122
code (digi.xbee.models.mode.IPAddressingMode attribute), 122
code (digi.xbee.models.mode.NeighborDiscoveryMode attribute), 123
code (digi.xbee.models.mode.OperatingMode attribute), 121
code (digi.xbee.models.options.DiscoveryOptions attribute), 134
code (digi.xbee.models.options.RegisterKeyOptions attribute), 135
code (digi.xbee.models.options.SendDataRequestOptions attribute), 136

code (digi.xbee.models.options.SocketOption attribute), 134
code (digi.xbee.models.options.XBeeLocalInterface attribute), 135
code (digi.xbee.models.protocol.IPProtocol attribute), 136

code (digi.xbee.models.protocol.XBeeProtocol attribute), 136

code (digi.xbee.models.status.AssociationIndicationStatus attribute), 142
code (digi.xbee.models.status.ATCommandStatus attribute), 143
code (digi.xbee.models.status.CellularAssociationIndicationStatus attribute), 143
code (digi.xbee.models.status.DeviceCloudStatus attribute), 143
code (digi.xbee.models.status.DiscoveryStatus attribute), 138
code (digi.xbee.models.status.FrameError attribute), 144
code (digi.xbee.models.status.ModemStatus attribute), 144
command (digi.xbee.packets.common.RemoteATCommandResponsePacket
attribute), 179
command (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket
attribute), 308
command (digi.xbee.packets.wifi.RemoteATCommandWifiPacket
attribute), 304
command_value (digi.xbee.packets.common.ATCommResponsePacket
attribute), 170
command_value (digi.xbee.packets.common.RemoteATCommandResponsePacket
attribute), 179
command_value (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket
attribute), 308
CommunicationException, 723
connect () (digi.xbee.filesystem.LocalXBeeFileSystemManager
method), 727
copy () (digi.xbee.reader.RouteInformationReceived
method), 758
copy () (digi.xbee.reader.RouteRecordIndicatorReceived
method), 759
copy () (digi.xbee.reader.SMSReceived method), 751
copy () (digi.xbee.reader.SocketDataReceived method), 750
copy () (digi.xbee.reader.SocketDataReceivedFrom
method), 754
copy () (digi.xbee.reader.SocketStateReceived method), 754
copy () (digi.xbee.reader.XBeeEvent method), 742
copy () (digi.xbee.reader.XBeeEvent method), 742
copy () (digi.xbee.reader.bluetooth.BluetoothDataReceived
method), 752
copy () (digi.xbee.reader.bluetooth.DataReceived method), 751
copy () (digi.xbee.reader.bluetooth.DeviceDiscovered method), 747
copy () (digi.xbee.reader.bluetooth.DiscoveryProcessFinished
method), 748
copy () (digi.xbee.reader.bluetooth.SocketDataReceivedFrom
method), 748
copy () (digi.xbee.reader.bluetooth.SocketStateReceived
method), 754
copy () (digi.xbee.reader.bluetooth.SocketStateReceived
method), 754
copy () (digi.xbee.reader.bluetooth.SocketStateTransferred
method), 754
copy () (digi.xbee.reader.bluetooth.SocketStateTransferred
method), 754
Connection (class in digi.xbee.devices), 721ConnectionException, 723
content_type (digi.xbee.packets.devicecloud.SendDataRequestPacket
attribute), 213
COORDINATOR_ADDRESS
(digi.xbee.models.address.XBee16BitAddress
attribute), 124
COORDINATOR_ADDRESS
(digi.xbee.models.address.XBee64BitAddress
attribute), 125
copy () (digi.xbee.reader.bluetooth.BluetoothDataReceived
method), 752
copy () (digi.xbee.reader.bluetooth.DataReceived method), 744
copy () (digi.xbee.reader.bluetooth.DeviceDiscovered method), 747
copy () (digi.xbee.reader.bluetooth.DiscoveryProcessFinished
method), 748
copy () (digi.xbee.reader.bluetooth.EndDiscoveryScan method), 743
copy () (digi.xbee.reader.bluetooth.ExplicitDataReceived
method), 749
copy () (digi.xbee.reader.bluetooth.InitDiscoveryScan method), 759
copy () (digi.xbee.reader.bluetooth.IOSampleReceived method), 746
copy () (digi.xbee.reader.bluetooth.IPDataReceived method), 750
copy () (digi.xbee.reader.bluetooth.MicroPythonDataReceived
method), 753
copy () (digi.xbee.reader.bluetooth.ModemStatusReceived
method), 745
copy () (digi.xbee.reader.bluetooth.NetworkModified method), 747
copy () (digi.xbee.reader.bluetooth.PacketReceived method), 743
copy () (digi.xbee.reader.bluetooth.RelayDataReceived
method), 749
copy () (digi.xbee.reader.bluetooth.SocketDataReceived
method), 751
copy () (digi.xbee.reader.bluetooth.SocketStateReceived
method), 755
copy () (digi.xbee.reader.bluetooth.SocketStateTransferred
method), 756
copy () (digi.xbee.reader.bluetooth.SocketStateTransferred
method), 754
Index
793
create_packet() (\texttt{digi.xbee.packets.base.GenericXBeePacket})
static method, 153
create_packet() (\texttt{digi.xbee.packets.base.UnknownXBeePacket})
static method, 155
create_packet() (\texttt{digi.xbee.packets.base.XBeeAPIOPacket})
static method, 151
create_packet() (\texttt{digi.xbee.packets.cellular.RXSMSPacket})
static method, 149
create_packet() (\texttt{digi.xbee.packets.cellular.TXSMSPacket})
static method, 158
create_packet() (\texttt{digi.xbee.packets.common.IODataSampleRxIndicatorPacket})
static method, 160
create_packet() (\texttt{digi.xbee.packets.common.IODataSampleRxIndicatorWifiPacket})
static method, 166
create_packet() (\texttt{digi.xbee.packets.common.ModemStatusPacket})
static method, 168
create_packet() (\texttt{digi.xbee.packets.common.ReceivePacket})
static method, 172
create_packet() (\texttt{digi.xbee.packets.common.RemoteATCommandPacket})
static method, 175
create_packet() (\texttt{digi.xbee.packets.common.RemoteATCommandResponsePacket})
static method, 178
create_packet() (\texttt{digi.xbee.packets.common.TransmitPacket})
static method, 182
create_packet() (\texttt{digi.xbee.packets.common.TransmitStatusPacket})
static method, 185
create_packet() (\texttt{digi.xbee.packets.devicelcloud.DeviceRequestPacket})
static method, 201
create_packet() (\texttt{digi.xbee.packets.devicelcloud.DeviceResponsePacket})
static method, 204
create_packet() (\texttt{digi.xbee.packets.devicelcloud.DeviceStatusPacket})
static method, 207
create_packet() (\texttt{digi.xbee.packets.devicelcloud.FrameErrorPacket})
static method, 210
create_packet() (\texttt{digi.xbee.packets.devicelcloud.SendDataRequestPacket})
static method, 212
create_packet() (\texttt{digi.xbee.packets.devicelcloud.SendDataResponsePacket})
static method, 216
create_packet() (\texttt{digi.xbee.packets.digimesh.RouteInformationPacket})
static method, 218
create_packet() (\texttt{digi.xbee.packets.network.RXIPv4Packet})
static method, 223
create_packet() (\texttt{digi.xbee.packets.network.TXIPv4Packet})
static method, 226
create_packet() (\texttt{digi.xbee.packets.raw.TX16Packet})
static method, 229
create_packet() (\texttt{digi.xbee.packets.raw.TXStatusPacket})
static method, 234
create_packet() (\texttt{digi.xbee.packets.raw.TX16IOPacket})
static method, 237
create_packet() (\texttt{digi.xbee.packets.raw.TX64Packet})
static method, 240
create_packet() (\texttt{digi.xbee.packets.raw.TX64IOPacket})
static method, 243
create_packet() (\texttt{digi.xbee.packets.socket.ReceivePacket})
static method, 249
create_packet() (\texttt{digi.xbee.packets.socket.SocketCreatePacket})
static method, 254
create_packet() (\texttt{digi.xbee.packets.socket.SocketCreateResponsePacket})
static method, 257
create_packet() (\texttt{digi.xbee.packets.socket.SocketListenResponsePacket})
static method, 266
create_packet() (\texttt{digi.xbee.packets.socket.SocketListenResponsePacket})
static method, 269
create_packet() (\texttt{digi.xbee.packets.socket.SocketCreateResponsePacket})
static method, 274
create_packet() (\texttt{digi.xbee.packets.raw.TX64Packet})
static method, 278
create_packet() (\texttt{digi.xbee.packets.socket.SocketNewIPv4ClientPacket})
static method, 283
create_packet() (\texttt{digi.xbee.packets.socket.SocketClosePacket})
static method, 288
create_packet() (\texttt{digi.xbee.packets.socket.SocketClosePacket})
static method, 291
create_packet() (\texttt{digi.xbee.packets.socket.SocketSendPacket})
static method, 295
create_packet() (\texttt{digi.xbee.packets.socket.SocketSendToPacket})
static method, 298
create_packet() (\texttt{digi.xbee.packets.socket.SocketSendPacket})
static method, 306
create_packet() (digi.xbee.packets.wifi.RemoteATCommandWifiPacket static method), 303
create_packet() (digi.xbee.packets.zigbee.RegisterDeviceStatusPacket static method), 312
create_packet() (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket static method), 309
create_packet() (digi.xbee.packets.zigbee.RouteRecordIndicatorPacket static method), 315
create_xbee_device() (digi.xbee.devices.CellularDevice class method), 498
create_xbee_device() (digi.xbee.devices.DigiMeshDevice class method), 392
create_xbee_device() (digi.xbee.devices.DigiPointDevice class method), 417
create_xbee_device() (digi.xbee.devices.IPDevice class method), 475
create_xbee_device() (digi.xbee.devices.LPWANDevice class method), 520
create_xbee_device() (digi.xbee.devices.NBIoTDevice class method), 544
create_xbee_device() (digi.xbee.devices.Raw802Device class method), 366
create_xbee_device() (digi.xbee.devices.WiFiDevice class method), 574
create_xbee_device() (digi.xbee.devices.XBeeDevice class method), 339
create_xbee_device() (digi.xbee.devices.ZigBeeDevice class method), 447

data (digi.xbee.packets.cellular.RXSMSPacket attribute), 158
data (digi.xbee.packets.cellular.TXSMSPacket attribute), 163
data (digi.xbee.packets.network.RXIPv4Packet attribute), 223
data (digi.xbee.packets.network.TXIPv4Packet attribute), 228
DataReceived (class in digi.xbee.reader), 743
DEFAULT_TIME_BETWEEN_REQUESTS (digi.xbee.devices.XBeeNetwork attribute), 672
DEFAULT_TIME_BETWEEN_SCANS (digi.xbee.devices.XBeeNetwork attribute), 671
del_bluetooth_data_received_callback() (digi.xbee.devices.CellularDevice method), 499
del_bluetooth_data_received_callback() (digi.xbee.devices.DigiMeshDevice method), 392
del_bluetooth_data_received_callback() (digi.xbee.devices.DigiPointDevice method), 417
del_bluetooth_data_received_callback() (digi.xbee.devices.IPDevice method), 476
del_bluetooth_data_received_callback() (digi.xbee.devices.LPWANDevice method), 521
del_bluetooth_data_received_callback() (digi.xbee.devices.NBIoTDevice method), 545
del_bluetooth_data_received_callback() (digi.xbee.devices.Raw802Device method), 367
del_bluetooth_data_received_callback() (digi.xbee.devices.WiFiDevice method), 574
del_bluetooth_data_received_callback() (digi.xbee.devices.XBeeDevice method), 346
del_bluetooth_data_received_callback() (digi.xbee.devices.ZigBeeDevice method), 448
del_bluetooth_data_received_callback() (digi.xbee.reader.PacketListener method), 766
del_data_received_callback() (digi.xbee.devices.CellularDevice method), 499
del_data_received_callback() (digi.xbee.devices.DigiMeshDevice method), 393
del_data_received_callback() (digi.xbee.devices.DigiPointDevice method), 417
del_data_received_callback() (digi.xbee.devices.IPDevice method), 472
del_data_received_callback() (digi.xbee.devices.LPWANDevice method),
del_data_received_callback() (digi.xbee.devices.NBIoTDevice method), 693

del_data_received_callback() (digi.xbee.devices.XBeeNetwork method), 675

del_data_received_callback() (digi.xbee.devices.ZigBeeNetwork method), 684

del_explicit_data_received_callback() (digi.xbee.devices.CellularDevice method), 521

del_explicit_data_received_callback() (digi.xbee.devices.DigiMeshDevice method), 499

del_explicit_data_received_callback() (digi.xbee.devices.DigiPointDevice method), 418

del_explicit_data_received_callback() (digi.xbee.devices.IPDevice method), 472

del_explicit_data_received_callback() (digi.xbee.devices.LPWANDevice method), 521

del_explicit_data_received_callback() (digi.xbee.devices.Raw802Device method), 367

del_explicit_data_received_callback() (digi.xbee.devices.XBeeDevice method), 545

del_explicit_data_received_callback() (digi.xbee.devices.ZigBeeDevice method), 346

del_explicit_data_received_callback() (digi.xbee.reader.PacketListener method), 765

del_init_discovery_scan_callback() (digi.xbee.devices.DigiMeshNetwork method), 703

del_init_discovery_scan_callback() (digi.xbee.devices.DigiPointNetwork method), 712

del_init_discovery_scan_callback() (digi.xbee.devices.XBeeNetwork method), 703

del_init_discovery_scan_callback() (digi.xbee.devices.ZigBeeNetwork method), 675

del_init_discovery_scan_callback() (digi.xbee.devices.Raw802Network method), 684

del_io_sample_received_callback() (digi.xbee.devices.CellularDevice method), 796

Index
del_io_sample_received_callback()
  (digi.xbee.devices.DigiMeshDevice method), 393

del_io_sample_received_callback()
  (digi.xbee.devices.DigiPointDevice method), 418

del_io_sample_received_callback()
  (digi.xbee.devices.IPDevice method), 476

del_io_sample_received_callback()
  (digi.xbee.devices.LPWANDevice method), 521

del_io_sample_received_callback()
  (digi.xbee.devices.NBIoTDevice method), 545

del_io_sample_received_callback()
  (digi.xbee.devices.Raw802Device method), 575

del_io_sample_received_callback()
  (digi.xbee.devices.XBeeDevice method), 346

del_io_sample_received_callback()
  (digi.xbee.devices.ZigBeeDevice method), 448

del_ip_data_received_callback()
  (digi.xbee.devices.CellularDevice method), 499

del_ip_data_received_callback()
  (digi.xbee.devices.IPDevice method), 469

del_ip_data_received_callback()
  (digi.xbee.devices.LPWANDevice method), 521

del_ip_data_received_callback()
  (digi.xbee.devices.NBIoTDevice method), 545

del_ip_data_received_callback()
  (digi.xbee.devices.WiFiDevice method), 575

del_ip_data_received_callback()
  (digi.xbee.reader.PacketListener method), 765

del_micropython_data_received_callback()
  (digi.xbee.devices.CellularDevice method), 499

del_micropython_data_received_callback()
  (digi.xbee.devices.XBeeDevice method), 418

del_micropython_data_received_callback()
  (digi.xbee.devices.ZigBeeDevice method), 448

del_modem_status_received_callback()
  (digi.xbee.devices.CellularDevice method), 499

del_modem_status_received_callback()
  (digi.xbee.devices.DigiMeshDevice method), 393

del_modem_status_received_callback()
  (digi.xbee.devices.DigiPointDevice method), 418

del_modem_status_received_callback()
  (digi.xbee.devices.IPDevice method), 476

del_modem_status_received_callback()
  (digi.xbee.devices.LPWANDevice method), 522

del_modem_status_received_callback()
  (digi.xbee.devices.NBIoTDevice method), 545

del_modem_status_received_callback()
  (digi.xbee.devices.WiFiDevice method), 575

del_modem_status_received_callback()
  (digi.xbee.devices.XBeeDevice method), 346

del_modem_status_received_callback()
  (digi.xbee.devices.ZigBeeDevice method), 449

del_modem_status_received_callback()
  (digi.xbee.reader.PacketListener method), 766

del_network_modified_callback()
  (digi.xbee.devices.DigiMeshDevice method), 703

del_network_modified_callback()
  (digi.xbee.devices.DigiPointNetwork method), 713

del_network_modified_callback()
  (digi.xbee.devices.XBeeNetwork method), 675

del_network_modified_callback()
  (digi.xbee.devices.ZigBeeNetwork method), 684
del_socket_data_received_from_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.IPDevice method), 477 (digi.xbee.devices.NBIoTDevice method), 546
del_socket_data_received_from_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.LPWANDevice method), 522 (digi.xbee.devices.Raw802Device method), 368
del_socket_data_received_from_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.NBIoTDevice method), 546 (digi.xbee.devices.WiFiDevice method), 576
del_socket_data_received_from_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.Raw802Device method), 368 (digi.xbee.devices.ZigBeeDevice method), 449
del_socket_data_received_from_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.WiFiDevice method), 576 (digi.xbee.devices.XBeeDevice method), 347
del_socket_data_received_from_callback() deprecated() (in module digi.xbee.util.utils), 322
del_socket_data_received_from_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 449 (digi.xbee.reader.PacketListener method), 765
del_socket_state_callback()
(digi.xbee.xsocket.socket method), 778
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.CellularDevice method), 500 (digi.xbee.devices.XBeeDevice method), 779
del_socket_state_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 394
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.DigiPointDevice method), 419 (digi.xbee.models.accesspoint.WiFiEncryptionType attribute), 115
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.IPDevice method), 477 (digi.xbee.models.option.DiscoveryOptions attribute), 134
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.LPWANDevice method), 522 (digi.xbee.models.option.SendDataRequestOptions attribute), 133
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.WiFiDevice method), 576 (digi.xbee.models.option.SocketOption attribute), 122
del_socket_state_received_callback()
(digi.xbee.devices.XBeeDevice method), 346

del_socket_state_received()
(digi.xbee.xsocket.socket method), 778
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.CellularDevice method), 500
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 394
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.DigiPointDevice method), 419
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.IPDevice method), 477
del_socket_state_received_callback() del_user_data_relay_received_callback()
(digi.xbee.devices.LPWANDevice method), 522
description (digi.xbee.models.status.CellularAssociationIndicationStatus attribute), 127
dest_endpoint (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 197
dest_endpoint (digi.xbee.packets.common.ExplicitRXIndicatorPacket attribute), 200
dest_interface (digi.xbee.packets.socket.SocketConnectPacket attribute), 249
dest_port (digi.xbee.models.message.IPMessage attribute), 129
dest_port (digi.xbee.packets.network.RXIPv4Packet attribute), 223
dest_port (digi.xbee.packets.network.TXIPv4Packet attribute), 228
dest_port (digi.xbee.packets.socket.SocketSendToPacket attribute), 281
DeviceCloudStatus (class in digi.xbee.models.status), 143
DeviceDiscovered (class in digi.xbee.reader), 747
DeviceRequestPacket (class in digi.xbee.packets.devicecloud), 201
DeviceResponseStatusPacket (class in digi.xbee.packets.devicecloud), 206
DictKeys (class in digi.xbee.packets.base), 149
digi (module), 114
digi.xbee (module), 114
digi.xbee.comm_interface (module), 322
digi.xbee.devices (module), 323
digi.xbee.exception (module), 723
digi.xbee.filesystem (module), 725
digi.xbee.firmware (module), 729
digi.xbee.io (module), 730
digi.xbee.models (module), 114
digi.xbee.models.accesspoint (module), 114
digi.xbee.models.address (module), 123
digi.xbee.models.atcommand (module), 115
digi.xbee.models.hw (module), 119
digi.xbee.models.message (module), 126
digi.xbee.models.mode (module), 121
digi.xbee.models.options (module), 130
digi.xbee.models.protocol (module), 135
digi.xbee.models.status (module), 137
digi.xbee.packets (module), 147
digi.xbee.packets.aft (module), 147
digi.xbee.packets.base (module), 147
digi.xbee.packets.cellular (module), 157
digi.xbee.packets.common (module), 163
digi.xbee.packets.devicecloud (module), 201
digi.xbee.packets.digimesh (module), 217
digi.xbee.packets.factory (module), 318
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>digi.xbee.packets.network</td>
<td>222</td>
</tr>
<tr>
<td>digi.xbee.packets.raw</td>
<td>228</td>
</tr>
<tr>
<td>digi.xbee.packets.relay</td>
<td>248</td>
</tr>
<tr>
<td>digi.xbee.packets.socket</td>
<td>254</td>
</tr>
<tr>
<td>digi.xbee.packets.wifi</td>
<td>299</td>
</tr>
<tr>
<td>digi.xbee.packets.zigbee</td>
<td>308</td>
</tr>
<tr>
<td>digi.xbee.profile</td>
<td>735</td>
</tr>
<tr>
<td>digi.xbee.reader</td>
<td>742</td>
</tr>
<tr>
<td>digi.xbee.recovery</td>
<td>771</td>
</tr>
<tr>
<td>digi.xbee.serial</td>
<td>772</td>
</tr>
<tr>
<td>digi.xbee.util</td>
<td>318</td>
</tr>
<tr>
<td>digi.xbee.util.utils</td>
<td>318</td>
</tr>
<tr>
<td>digi.xbee.xsocket</td>
<td>774</td>
</tr>
<tr>
<td>DIGIMESH_MODE (attribute)</td>
<td>132</td>
</tr>
<tr>
<td>DigiMeshDevice (class)</td>
<td>386</td>
</tr>
<tr>
<td>DigiMeshNetwork (class)</td>
<td>700</td>
</tr>
<tr>
<td>DigiPointDevice (class)</td>
<td>412</td>
</tr>
<tr>
<td>DigiPointNetwork (class)</td>
<td>709</td>
</tr>
<tr>
<td>digital_hsb_mask (attribute)</td>
<td>734</td>
</tr>
<tr>
<td>DIGITAL_IN (attribute)</td>
<td>735</td>
</tr>
<tr>
<td>digital_lsb_mask (attribute)</td>
<td>734</td>
</tr>
<tr>
<td>digital_mask (attribute)</td>
<td>734</td>
</tr>
<tr>
<td>DIGITAL_OUT_HIGH (attribute)</td>
<td>735</td>
</tr>
<tr>
<td>DIGITAL_OUT_LOW (attribute)</td>
<td>735</td>
</tr>
<tr>
<td>digital_values (attribute)</td>
<td>734</td>
</tr>
<tr>
<td>DISABLE_ACK (attribute)</td>
<td>133</td>
</tr>
<tr>
<td>DISABLED (attribute)</td>
<td>734</td>
</tr>
<tr>
<td>disable_bluetooth() (AbstractXBeeDevice method)</td>
<td>336</td>
</tr>
<tr>
<td>disable_bluetooth() (CellularDevice method)</td>
<td>500</td>
</tr>
<tr>
<td>disable_bluetooth() (DigiMeshDevice method)</td>
<td>394</td>
</tr>
<tr>
<td>disable_bluetooth() (DigiPointDevice method)</td>
<td>419</td>
</tr>
<tr>
<td>disable_bluetooth() (IPDevice method)</td>
<td>477</td>
</tr>
<tr>
<td>disable_bluetooth() (LPWANDevice method)</td>
<td>522</td>
</tr>
<tr>
<td>disable_bluetooth() (Raw802Device method)</td>
<td>368</td>
</tr>
<tr>
<td>disable_bluetooth() (RemoteDigiMeshDevice method)</td>
<td>626</td>
</tr>
<tr>
<td>disable_bluetooth() (RemoteDigiPointDevice method)</td>
<td>641</td>
</tr>
<tr>
<td>disable_bluetooth() (RemoteRaw802Device method)</td>
<td>611</td>
</tr>
<tr>
<td>disable_bluetooth() (RemoteXBeeDevice method)</td>
<td>597</td>
</tr>
<tr>
<td>disable_bluetooth() (RemoteZigBeeDevice method)</td>
<td>657</td>
</tr>
<tr>
<td>disable_bluetooth() (WiFiDevice method)</td>
<td>576</td>
</tr>
<tr>
<td>disable_bluetooth() (XBeeDevice method)</td>
<td>349</td>
</tr>
<tr>
<td>disable_bluetooth() (ZigBeeDevice method)</td>
<td>449</td>
</tr>
<tr>
<td>disable_logger() (in module digi.xbee.util.utils)</td>
<td>322</td>
</tr>
<tr>
<td>disconnect() (AbstractXBeeDevice method)</td>
<td>727</td>
</tr>
<tr>
<td>disconnect() (CellularDevice method)</td>
<td>703</td>
</tr>
<tr>
<td>disconnect() (DigiMeshNetwork method)</td>
<td>713</td>
</tr>
<tr>
<td>disconnect() (DigiPointNetwork method)</td>
<td>694</td>
</tr>
<tr>
<td>disconnect() (IPDevice method)</td>
<td>673</td>
</tr>
<tr>
<td>disconnect() (LPWANDevice method)</td>
<td>684</td>
</tr>
<tr>
<td>disconnect() (Raw802Device method)</td>
<td>611</td>
</tr>
<tr>
<td>disconnect() (RemoteDigiMeshDevice method)</td>
<td>626</td>
</tr>
<tr>
<td>disconnect() (RemoteDigiPointDevice method)</td>
<td>641</td>
</tr>
<tr>
<td>disconnect() (RemoteRaw802Device method)</td>
<td>611</td>
</tr>
<tr>
<td>disconnect() (RemoteXBeeDevice method)</td>
<td>597</td>
</tr>
<tr>
<td>disconnect() (RemoteZigBeeDevice method)</td>
<td>657</td>
</tr>
<tr>
<td>disconnect() (WiFiDevice method)</td>
<td>576</td>
</tr>
<tr>
<td>disconnect() (XBeeDevice method)</td>
<td>349</td>
</tr>
<tr>
<td>disconnect() (ZigBeeDevice method)</td>
<td>449</td>
</tr>
</tbody>
</table>
704

discover_devices()
(digi.xbee.devices.DigiPointNetwork method), 713

discover_devices()
(digi.xbee.devices.Raw802Network method), 694

discover_devices()
(digi.xbee.devices.XBeeNetwork method), 673

discover_devices()
(digi.xbee.devices.ZigBeeNetwork method), 685

DISCOVER_MYSELF
(digi.xbee.models.options.DiscoveryOptions attribute), 134

discovery_status
(digi.xbee.packets.common.TransmitStatusPacket attribute), 186

DiscoveryOptions
(class in digi.xbee.models.options), 133

DiscoveryProcessFinished
(class in digi.xbee.reader), 748

DiscoveryStatus
(class in digi.xbee.models.status), 138

doc_enum()
(in module digi.xbee.util.utils), 322

DONT_ATTEMPT_RD
(digi.xbee.models.options.TransmitOptions attribute), 131

dst_addr
(digi.xbee.packets.digimesh.RouteInformationPacket attribute), 221

ENABLE_APS_ENCRYPTION
(digi.xbee.models.options.TransmitOptions attribute), 132

empty()
(digi.xbee.reader.XBeeQueue method), 770

enable_apply_changes()
(digi.xbee.devices.RemoteDigiMeshDevice method), 627

enable_apply_changes()
(digi.xbee.devices.RemoteDigiPointDevice method), 641

enable_apply_changes()
(digi.xbee.devices.RemoteRaw802Device method), 611

enable_apply_changes()
(digi.xbee.devices.RemoteXBeeDevice method), 597

enable_apply_changes()
(digi.xbee.devices.RemoteZigBeeDevice method), 658

enable_apply_changes()
(digi.xbee.devices.WiFiDevice method), 576

enable_apply_changes()
(digi.xbee.devices.XBeeDevice method), 349

enable_apply_changes()
(digi.xbee.devices.ZigBeeDevice method), 450

enable_apply_changes()
(digi.xbee.devices.AbstractXBeeDevice method), 336

enable_apply_changes()
(digi.xbee.devices.CellularDevice method), 500

enable_apply_changes()
(digi.xbee.devices.DigiMeshDevice method), 394

enable_apply_changes()
(digi.xbee.devices.DigiMeshDevice method), 394

enable_apply_changes()
(digi.xbee.devices.DigiPointDevice method), 419

enable_apply_changes()
(digi.xbee.devices.IPDevice method), 477

enable_apply_changes()
(digi.xbee.devices.LPWANDevice method), 523

enable_apply_changes()
(digi.xbee.devices.LPWANDevice method), 523

enable_apply_changes()
(digi.xbee.devices.NBIoTDevice method), 547

enable_apply_changes()
(digi.xbee.devices.Raw802Device method), 368

enable_apply_changes()
(digi.xbee.devices.Raw802Device method), 368

enable_apply_changes()
(digi.xbee.devices.NBIoTDevice method), 546

enable_apply_changes()
(digi.xbee.devices.RemoteDigiMeshDevice method), 627

enable_apply_changes()
(digi.xbee.devices.RemoteDigiPointDevice method), 611

enable_apply_changes()
(digi.xbee.devices.RemoteRaw802Device method), 611

enable_apply_changes()
(digi.xbee.devices.RemoteXBeeDevice method), 597

enable_apply_changes()
(digi.xbee.devices.RemoteZigBeeDevice method), 658

enable_apply_changes()
(digi.xbee.devices.WiFiDevice method), 576

enable_apply_changes()
(digi.xbee.devices.XBeeDevice method), 349

enable_apply_changes()
(digi.xbee.devices.ZigBeeDevice method), 450

enable_apply_changes()
(digi.xbee.devices.AbstractXBeeDevice method), 336

enable_apply_changes()
(digi.xbee.devices.CellularDevice method), 500

enable_apply_changes()
(digi.xbee.devices.DigiMeshDevice method), 394

enable_apply_changes()
(digi.xbee.devices.DigiPointDevice method), 419

enable_apply_changes()
(digi.xbee.devices.IPDevice method), 477

enable_apply_changes()
(digi.xbee.devices.LPWANDevice method), 523

enable_apply_changes()
(digi.xbee.devices.LPWANDevice method), 523

enable_apply_changes()
(digi.xbee.devices.NBIoTDevice method), 547

enable_apply_changes()
(digi.xbee.devices.Raw802Device method), 368

enable_apply_changes()
(digi.xbee.devices.RemoteDigiMeshDevice method), 627

enable_apply_changes()
(digi.xbee.devices.RemoteDigiPointDevice method), 611

enable_apply_changes()
(digi.xbee.devices.RemoteRaw802Device method), 611

enable_apply_changes()
(digi.xbee.devices.RemoteXBeeDevice method), 597

enable_apply_changes()
(digi.xbee.devices.RemoteZigBeeDevice method), 658

enable_apply_changes()
(digi.xbee.devices.WiFiDevice method), 576

enable_apply_changes()
(digi.xbee.devices.XBeeDevice method), 349

enable_apply_changes()
(digi.xbee.devices.ZigBeeDevice method), 450

enable_apply_changes()
(digi.xbee.devices.AbstractXBeeDevice method), 336
enable_bluetooth()
(digi.xbee.devices.RemoteRaw802Device method), 611

enable_bluetooth()
(digi.xbee.devices.RemoteDigiMeshDevice method), 627

execute_command()
(digi.xbee.devices.RemoteDigiMeshDevice method), 627

execute_command()
(digi.xbee.devices.RemoteDigiPointDevice method), 641

execute_command()
(digi.xbee.devices.RemoteRaw802Device method), 611

execute_command()
(digi.xbee.devices.RemoteXBeeDevice method), 597

execute_command()
(digi.xbee.devices.RemoteZigBeeDevice method), 658

execute_command()
(digi.xbee.devices.WiFiDevice method), 576

enable_bluetooth()
(digi.xbee.devices.XBeeDevice method), 350

enable_bluetooth()
(digi.xbee.devices.ZigBeeDevice method), 450

enable_logger() (in module digi.xbee.util.utils), 322

ENABLE_MULTICAST (digi.xbee.models.options.TransmitOptions attribute), 132

execute_command()
(digi.xbee.devices.Raw802Device method), 369

execute_command()
(digi.xbee.devices.RemoteDigiMeshDevice method), 627

execute_command()
(digi.xbee.devices.RemoteDigiPointDevice method), 641

execute_command()
(digi.xbee.devices.RemoteRaw802Device method), 611

execute_command()
(digi.xbee.devices.RemoteXBeeDevice method), 597

execute_command()
(digi.xbee.devices.RemoteZigBeeDevice method), 658

execute_command()
(digi.xbee.devices.WiFiDevice method), 576

enable_bluetooth()
(digi.xbee.devices.XBeeDevice method), 350

enable_logger() (in module digi.xbee.util.utils), 322

ENABLE_MULTICAST (digi.xbee.models.options.TransmitOptions attribute), 132

execute_command()
(digi.xbee.devices.Raw802Device method), 369

execute_command()
(digi.xbee.devices.RemoteDigiMeshDevice method), 627

execute_command()
(digi.xbee.devices.RemoteDigiPointDevice method), 641

execute_command()
(digi.xbee.devices.RemoteRaw802Device method), 611

execute_command()
(digi.xbee.devices.RemoteXBeeDevice method), 597

execute_command()
(digi.xbee.devices.RemoteZigBeeDevice method), 658

execute_command()
(digi.xbee.devices.WiFiDevice method), 576

encrypt_type (digi.xbee.models.accesspoint.AccessPoint attribute), 115

EndDiscoveryScan (class in digi.xbee.reader), 760

ExplicitAddressingPacket (class in digi.xbee.packets.common), 193

ExplicitDataReceived (class in digi.xbee.reader), 749

ExplicitRXIndicatorPacket (class in digi.xbee.packets.common), 197

ExplicitXBeeMessage (class in digi.xbee.models.message), 127

extend() (digi.xbee.reader.BluetoothDataReceived method), 752

extend() (digi.xbee.reader.DataReceived method), 744

extend() (digi.xbee.reader.DeviceDiscovered method), 747

extend() (digi.xbee.reader.DiscoveryProcessFinished method), 748

extend() (digi.xbee.reader.EndDiscoveryScan method), 760

extend() (digi.xbee.reader.ExplicitDataReceived method), 749

extend() (digi.xbee.reader.InitDiscoveryScan method), 760

extend() (digi.xbee.reader.IOSampleReceived method), 746

extend() (digi.xbee.reader.IPDataReceived method), 750

extend() (digi.xbee.reader.MicroPythonDataReceived method), 753

extend() (digi.xbee.reader.ModemStatusReceived method), 745
extend() (digi.xbee.reader.NetworkModified method), 747
extend() (digi.xbee.reader.PacketReceived method), 743
extend() (digi.xbee.reader.RelayDataReceived method), 752
extend() (digi.xbee.reader.RouteInformationReceived method), 758
extend() (digi.xbee.reader.RouteReceived method), 759
extend() (digi.xbee.reader.RouteRecordIndicatorReceived method), 757
extend() (digi.xbee.reader.RouteRecordReceived method), 755
extend() (digi.xbee.reader.SMSReceived method), 751
extend() (digi.xbee.reader.SocketDataReceived method), 756
extend() (digi.xbee.reader.SocketDataReceivedFrom method), 750
extend() (digi.xbee.reader.SocketStateReceived method), 754
extend() (digi.xbee.reader.XBeeEvent method), 742
EXTENDED_TIMEOUT (digi.xbee.models.options.RemoteATCmdOptions attribute), 133
flush_queues() (digi.xbee.devices.DigiMeshDevice method), 395
flush_queues() (digi.xbee.devices.DigiPointDevice method), 420
flush_queues() (digi.xbee.devices.IPDevice method), 478
flush_queues() (digi.xbee.devices.LPWANDevice method), 523
flush_queues() (digi.xbee.devices.NBIoTDevice method), 547
flush_queues() (digi.xbee.devices.Raw802Device method), 369
flush_queues() (digi.xbee.devices.WiFiDevice method), 577
flush_queues() (digi.xbee.devices.XBeeDevice method), 444
force_disassociate() (digi.xbee.devices.RemoteZigBeeDevice method), 655
force_disassociate() (digi.xbee.devices.ZigBeeDevice method), 438
format() (digi.xbee.profile.XBeeProfileSetting attribute), 739
format_filesystem() (digi.xbee.filesystem.LocalXBeeFileSystemManager method), 729
frame_id (digi.xbee.packets.base.GenericXBeePacket attribute), 153
frame_id (digi.xbee.packets.base.UnknownXBeePacket attribute), 156
frame_id (digi.xbee.packets.base.XBeeAPIPacket attribute), 151
frame_id (digi.xbee.packets.cellular.RXSMSPacket attribute), 158
frame_id (digi.xbee.packets.cellular.TXSMSPacket attribute), 161
frame_id (digi.xbee.packets.common.ATCommPacket attribute), 164
frame_id (digi.xbee.packets.common.ATCommQueuePacket attribute), 167
frame_id (digi.xbee.packets.common.ATCommResponsePacket attribute), 170
frame_id (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 195
frame_id (digi.xbee.packets.common.ExplicitRXIndicatorPacket attribute), 198
frame_id (digi.xbee.packets.common.IODataSampleRxIndicatorPacket attribute), 192
frame_id (digi.xbee.packets.common.ModemStatusPacket attribute), 189
frame_id (digi.xbee.packets.common.ReceivePacket attribute), 182
frame_id (digi.xbee.packets.common.RemoteATCommandPacket attribute), 176
frame_id (digi.xbee.packets.common.RemoteATCommandResponsePacket attribute), 255
frame_id (digi.xbee.packets.common.TransmitPacket frame_id (digi.xbee.packets.socket.SocketCreatePacket attribute), 209
attribute), 137
frame_id (digi.xbee.packets.common.TransmitStatusPacket frame_id (digi.xbee.packets.socket.SocketCreateResponsePacket attribute), 258
attribute), 183
frame_id (digi.xbee.packets.common.TransmitStatusPacket frame_id (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 286
attribute), 186
frame_id (digi.xbee.packets.devicecloud.DeviceRequestPacket frame_id (digi.xbee.packets.socket.SocketNewIPv4ClientPacket attribute), 289
attribute), 202
frame_id (digi.xbee.packets.devicecloud.DeviceResponsePacket frame_id (digi.xbee.packets.socket.SocketOptionRequestPacket attribute), 260
attribute), 205
frame_id (digi.xbee.packets.devicecloud.DeviceResponsePacket frame_id (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
attribute), 208
frame_id (digi.xbee.packets.devicecloud.FrameErrorPacket frame_id (digi.xbee.packets.socket.SocketReceiveFromPacket attribute), 294
attribute), 210
frame_id (digi.xbee.packets.devicecloud.FrameErrorPacket frame_id (digi.xbee.packets.socket.SocketReceivePacket attribute), 292
attribute), 213
frame_id (digi.xbee.packets.devicecloud.SendDataRequestPacket frame_id (digi.xbee.packets.socket.SocketSendPacket attribute), 278
attribute), 215
frame_id (digi.xbee.packets.devicecloud.SendDataResponsePacket frame_id (digi.xbee.packets.socket.SocketSendToPacket attribute), 281
attribute), 220
frame_id (digi.xbee.packets.devicecloud.RouteInformationPacket frame_id (digi.xbee.packets.socket.SocketStatePacket attribute), 297
attribute), 222
frame_id (digi.xbee.packets.network.RXIPv4Packet attribute), 223
frame_id (digi.xbee.packets.network.TXIPv4Packet attribute), 226
frame_id (digi.xbee.packets.raw.RX16IOPacket attribute), 246
frame_id (digi.xbee.packets.raw.RX16Packet attribute), 241
frame_id (digi.xbee.packets.raw.RX64IOPacket attribute), 244
frame_id (digi.xbee.packets.raw.RX64Packet attribute), 238
frame_id (digi.xbee.packets.raw.TX16Packet attribute), 232
frame_id (digi.xbee.packets.raw.TX64Packet attribute), 229
frame_id (digi.xbee.packets.raw.TXStatusPacket attribute), 235
frame_id (digi.xbee.packets.relay.UserDataRelayOutputPacket from_bytes() (digi.xbee.models.address.XBee16BitAddress class method), 124
attribute), 252
frame_id (digi.xbee.packets.relay.UserDataRelayPacket from_hex_string() (digi.xbee.models.address.XBee16BitAddress class method), 124
attribute), 249
frame_id (digi.xbee.packets.socket.SocketBindListenPacket from_hex_string() (digi.xbee.models.address.XBee64BitAddress class method), 125
attribute), 284
frame_id (digi.xbee.packets.socket.SocketClosePacket from_string() (digi.xbee.models.address.XBeeIMEIAddress class method), 125
attribute), 272
frame_id (digi.xbee.packets.socket.SocketCloseResponsePacket full() (digi.xbee.reader.XBeeQueue method), 771
attribute), 275
frame_id (digi.xbee.packets.socket.SocketConnectPacket attribute), 267
frame_id (digi.xbee.packets.socket.SocketConnectResponsePacket GenericXBeepacket (class in
get () (digi.xbee.reader.XBeeQueue method), 769
get_16bit_addr () (digi.xbee.devices.AbstractXBeeDevice.get_64bit_addr() method), 527
get_64bit_addr () (digi.xbee.devices.RemoteRaw802Device method), 612
get_ai_status () (digi.xbee.devices.RemoteZigBeeDevice method), 450
get_access_point () (digi.xbee.devices.ZigBeeDevice method), 566
get_access_point_timeout () (digi.xbee.devices.ZigBeeDevice method), 568
get_adc_value () (digi.xbee.devices.AbstractXBeeDevice method), 331
get_access_point () (digi.xbee.devices.RemoteDigiPointDevice method), 501
get_access_point_timeout () (digi.xbee.devices.RemoteZigBee Device method), 420
get_access_point() (digi.xbee.devices.Raw802Device method), 658
get_access_point_timeout() (digi.xbee.devices.RemoteDigiMeshDevice method), 627
get_access_point_timeout() (digi.xbee.devices.DigiMeshDevice method), 642
get_access_point_timeout() (digi.xbee.devices.AbstractXBeeDevice method), 640
get_access_point_timeout() (digi.xbee.devices.CellularDevice method), 519
get_adc_value () (digi.xbee.devices.RemoteDigiPointDevice method), 400
get_adc_value () (digi.xbee.devices.RemoteDigiMeshDevice method), 395
get_adc_value () (digi.xbee.devices.RemoteRaw802Device method), 612
get_adc_value () (digi.xbee.devices.ZigBeeDevice method), 420
get_adc_value () (digi.xbee.devices.IPDevice method), 478
get_adc_value () (digi.xbee.devices.LPWANDevice method), 477
get_adc_value () (digi.xbee.devices.RemoteDigiPointDevice method), 577
get_adc_value () (digi.xbee.devices.RemoteDigiMeshDevice method), 523
get_adc_value () (digi.xbee.devices.RemoteRaw802Device method), 642
get_adc_value () (digi.xbee.devices.RemoteDigiMeshDevice method), 655
get_adc_value () (digi.xbee.devices.AbstractXBeeDevice method), 375
get_adc_value () (digi.xbee.devices.ZigBeeDevice method), 400
get_ai_status () (digi.xbee.devices.ZigBeeDevice method), 550
get_ai_status () (digi.xbee.devices.RemoteDigiMeshDevice method), 627
get_ai_status () (digi.xbee.devices.RemoteZigBeeDevice method), 642
get_ai_status () (digi.xbee.devices.RemoteDigiMeshDevice method), 627
get_ai_status () (digi.xbee.devices.RemoteRaw802Device method), 642
Index
get_bluetooth_mac_addr()
(digi.xbee.devices.DigiPointDevice method), 421
get_bluetooth_mac_addr()
(digi.xbee.devices.IPDevice method), 479
get_bluetooth_mac_addr()
(digi.xbee.devices.LPWANDevice method), 525
get_bluetooth_mac_addr()
(digi.xbee.devices.NBIoTDevice method), 549
get_bluetooth_mac_addr()
(digi.xbee.devices.Raw802Device method), 371
get_bluetooth_mac_addr()
(digi.xbee.devices.RemoteDigiMeshDevice method), 629
get_bluetooth_mac_addr()
(digi.xbee.devices.RemoteDigiPointDevice method), 644
get_bluetooth_mac_addr()
(digi.xbee.devices.RemoteRaw802Device method), 613
get_bluetooth_mac_addr()
(digi.xbee.devices.RemoteXBeeDevice method), 599
get_bluetooth_mac_addr()
(digi.xbee.devices.RemoteZigBeeDevice method), 660
get_checksum()
(digi.xbee.packets.base.GenericXBeePacket method), 153
get_checksum()
(digi.xbee.packets.base.UnknownXBeePacket method), 156
get_checksum()
(digi.xbee.packets.base.XBeeAPIPacket method), 152
get_checksum()
(digi.xbee.packets.base.XBeePacket method), 149
get_checksum()
(digi.xbee.packets.cellular.RXSMSPacket method), 159
get_checksum()
(digi.xbee.packets.cellular.TXSMSPacket method), 161
get_checksum()
(digi.xbee.packets.common.ATCommPacket method), 164
get_checksum()
(digi.xbee.packets.common.ATCommQueuePacket method), 167
get_checksum()
(digi.xbee.packets.common.ATCommResponseStatusPacket method), 170
get_checksum()
(digi.xbee.packets.common.ExplicitAddressingPacket method), 196
get_checksum()
(digi.xbee.packets.common.ExplicitRXIndicatorPacket method), 198
get_checksum()
(digi.xbee.packets.common.IODataSampleRxIndicatorPacket method), 192
get_checksum()
(digi.xbee.packets.common.ModemStatusPacket method), 189
get_checksum()
(digi.xbee.packets.common.ReceivePacket method), 173
get_checksum()
(digi.xbee.packets.common.RemoteATCommandPacket method), 176
get_checksum()
(digi.xbee.packets.common.RemoteATCommandResponsePacket method), 180
get_checksum()
(digi.xbee.packets.common.TransmitPacket method), 183
get_checksum()
(digi.xbee.packets.common.TransmitStatusPacket method), 186
get_checksum()
(digi.xbee.packets.devicecloud.DeviceRequestPacket method), 202
get_checksum()
(digi.xbee.packets.devicecloud.DeviceResponsePacket method), 205
get_checksum()
(digi.xbee.packets.devicecloud.DeviceResponseStatusPacket method), 208
get_checksum()
(digi.xbee.packets.devicecloud.FrameErrorPacket method), 210
get_checksum()
(digi.xbee.packets.devicecloud.SendDataRequestPacket method), 213
get_checksum()
(digi.xbee.packets.devicecloud.SendDataResponsePacket method), 215
get_checksum()
(digi.xbee.packets.digimesh.RouteInformationPacket method), 220
get_checksum()
(digi.xbee.packets.network.RXIPv4Packet method), 223
get_checksum()
(digi.xbee.packets.network.TXIPv4Packet method), 226
get_checksum()
(digi.xbee.packets.raw.RX16IOPacket method), 246

Index 811

get_dest_ip_addr() (digi.xbee.devices.IPDevice method), 469
get_dest_ip_addr() (digi.xbee.devices.LPWANDevice method), 526
get_dest_ip_addr() (digi.xbee.devices.NBIoTDevice method), 549
get_dest_ip_addr() (digi.xbee.devices.WiFiDevice method), 579
get_device_by_16() (digi.xbee.devices.DigiMeshNetwork method), 705
get_device_by_16() (digi.xbee.devices.DigiPointNetwork method), 714
get_device_by_16() (digi.xbee.devices.Raw802Network method), 695
get_device_by_16() (digi.xbee.devices.XBeeNetwork method), 679
get_device_by_16() (digi.xbee.devices.ZigBeeNetwork method), 686
get_device_by_64() (digi.xbee.devices.DigiMeshNetwork method), 705
get_device_by_64() (digi.xbee.devices.DigiPointNetwork method), 714
get_device_by_64() (digi.xbee.devices.Raw802Network method), 695
get_device_by_64() (digi.xbee.devices.XBeeNetwork method), 679
get_device_by_64() (digi.xbee.devices.ZigBeeNetwork method), 686
get_discovery_callbacks() (digi.xbee.devices.DigiMeshNetwork method), 706
get_discovery_callbacks() (digi.xbee.devices.DigiPointNetwork method), 715
get_digital_value() (digi.xbee.io.IOSSample method), 733
get_dio_value() (digi.xbee.devices.AbstractXBeeDevice method), 333
get_dio_value() (digi.xbee.devices.CellularDevice method), 503
get_dio_value() (digi.xbee.devices.DigiMeshDevice method), 397
get_dio_value() (digi.xbee.devices.DigiPointDevice method), 422
get_dio_value() (digi.xbee.devices.IPDevice method), 480
get_dio_value() (digi.xbee.devices.LPWANDevice method), 526
get_dio_value() (digi.xbee.devices.NBIoTDevice method), 550
get_dio_value() (digi.xbee.devices.Raw802Device method), 371
get_dio_value() (digi.xbee.devices.RemoteDigiMeshDevice method), 630
get_dio_value() (digi.xbee.devices.RemoteDigiPointDevice method), 644
get_dio_value() (digi.xbee.devices.RemoteRaw802Device method), 614
get_dio_value() (digi.xbee.devices.RemoteXBeeDevice method), 600
get_dio_value() (digi.xbee.devices.RemoteZigBeeDevice method), 661
get_device_by_node_id() (digi.xbee.devices.DigiMeshNetwork method), 705
get_device_by_node_id() (digi.xbee.devices.DigiPointNetwork method), 715
get_device_by_node_id() (digi.xbee.devices.Raw802Network method), 696
get_device_by_node_id() (digi.xbee.devices.XBeeNetwork method), 679
get_device_by_node_id() (digi.xbee.devices.ZigBeeNetwork method), 686
get_devices() (digi.xbee.devices.DigiMeshNetwork method), 705
get_devices() (digi.xbee.devices.DigiPointNetwork method), 715
get_devices() (digi.xbee.devices.Raw802Network method), 696
get_devices() (digi.xbee.devices.XBeeNetwork method), 673
get_devices() (digi.xbee.devices.ZigBeeNetwork method), 686
get_digital_value() (digi.xbee.io.IOSSample method), 733
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_discovery_callbacks()</td>
<td>digi.xbee.devices.XBeeNetwork</td>
<td>method,</td>
<td>696</td>
</tr>
<tr>
<td>get_discovery_callbacks()</td>
<td>digi.xbee.devices.ZigBeeNetwork</td>
<td>method,</td>
<td>680</td>
</tr>
<tr>
<td>get_discovery_options()</td>
<td>digi.xbee.devices.DigiMeshNetwork</td>
<td>method,</td>
<td>687</td>
</tr>
<tr>
<td>get_discovery_options()</td>
<td>digi.xbee.devices.DigiPointNetwork</td>
<td>method,</td>
<td>706</td>
</tr>
<tr>
<td>get_discovery_options()</td>
<td>digi.xbee.devices.Raw802Network</td>
<td>method,</td>
<td>696</td>
</tr>
<tr>
<td>get_discovery_options()</td>
<td>digi.xbee.devices.XBeeNetwork</td>
<td>method,</td>
<td>676</td>
</tr>
<tr>
<td>get_discovery_options()</td>
<td>digi.xbee.devices.ZigBeeNetwork</td>
<td>method,</td>
<td>687</td>
</tr>
<tr>
<td>get_discovery_timeout()</td>
<td>digi.xbee.devices.DigiMeshNetwork</td>
<td>method,</td>
<td>706</td>
</tr>
<tr>
<td>get_discovery_timeout()</td>
<td>digi.xbee.devices.DigiPointNetwork</td>
<td>method,</td>
<td>715</td>
</tr>
<tr>
<td>get_discovery_timeout()</td>
<td>digi.xbee.devices.Raw802Network</td>
<td>method,</td>
<td>696</td>
</tr>
<tr>
<td>get_discovery_timeout()</td>
<td>digi.xbee.devices.XBeeNetwork</td>
<td>method,</td>
<td>677</td>
</tr>
<tr>
<td>get_discovery_timeout()</td>
<td>digi.xbee.devices.ZigBeeNetwork</td>
<td>method,</td>
<td>687</td>
</tr>
<tr>
<td>get_dns_address()</td>
<td>digi.xbee.devices.WiFiDevice</td>
<td>method,</td>
<td>571</td>
</tr>
<tr>
<td>get_explicit_data_received_callbacks()</td>
<td>digi.xbee.reader.PacketListener</td>
<td>method,</td>
<td>767</td>
</tr>
<tr>
<td>get_explicit_queue()</td>
<td>digi.xbee.reader.PacketListener</td>
<td>method,</td>
<td>762</td>
</tr>
<tr>
<td>get_file()</td>
<td>digi.xbee.filesystem.LocalXBeeFileManager</td>
<td>method,</td>
<td>728</td>
</tr>
<tr>
<td>get_file_hash()</td>
<td>digi.xbee.filesystem.LocalXBeeFileManager</td>
<td>method,</td>
<td>729</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.AbstractXBeeDevice</td>
<td>method,</td>
<td>327</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.CellularDevice</td>
<td>method,</td>
<td>504</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.DigiMeshDevice</td>
<td>method,</td>
<td>398</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.DigiPointDevice</td>
<td>method,</td>
<td>423</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.IPDevice</td>
<td>method,</td>
<td>526</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.LPWANDevice</td>
<td>method,</td>
<td>550</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.NBiotDevice</td>
<td>method,</td>
<td>580</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.RemoteDigiMeshDevice</td>
<td>method,</td>
<td>630</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.RemoteDigiPointDevice</td>
<td>method,</td>
<td>645</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.RemoteRaw802Device</td>
<td>method,</td>
<td>615</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.RemoteXBeeDevice</td>
<td>method,</td>
<td>601</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.RemoteZigBeeDevice</td>
<td>method,</td>
<td>661</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.WiFiDevice</td>
<td>method,</td>
<td>580</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.XBeeDevice</td>
<td>method,</td>
<td>353</td>
</tr>
<tr>
<td>get_firmware_version()</td>
<td>digi.xbee.devices.ZigBeeDevice</td>
<td>method,</td>
<td>453</td>
</tr>
<tr>
<td>get_frame_spec_data()</td>
<td>digi.xbee.packets.base.GenericXBeePacket</td>
<td>method,</td>
<td>154</td>
</tr>
<tr>
<td>get_frame_spec_data()</td>
<td>digi.xbee.packets.base.UnknownXBeePacket</td>
<td>method,</td>
<td>156</td>
</tr>
<tr>
<td>get_frame_spec_data()</td>
<td>digi.xbee.packets.base.XBeeAPacket</td>
<td>method,</td>
<td>150</td>
</tr>
<tr>
<td>get_frame_spec_data()</td>
<td>digi.xbee.packets.base.XBeePacket</td>
<td>method,</td>
<td>150</td>
</tr>
<tr>
<td>get_frame_spec_data()</td>
<td>digi.xbee.packets.cellular.RXSMSPacket</td>
<td>method,</td>
<td>150</td>
</tr>
</tbody>
</table>
get_frame_spec_data method, 159
get_frame_spec_data method, 214
get_frame_spec_data method, 246
get_frame_spec_data method, 241
get_frame_spec_data method, 238
get_frame_spec_data method, 235
get_frame_spec_data method, 233
get_frame_spec_data method, 230
get_frame_spec_data method, 227
get_frame_spec_data method, 224
get_frame_spec_data method, 220
get_frame_spec_data method, 198
get_frame_spec_data method, 196
get_frame_spec_data method, 192
get_frame_spec_data method, 189
get_frame_spec_data method, 187
get_frame_spec_data method, 185
get_frame_spec_data method, 183
get_frame_spec_data method, 181
get_frame_spec_data method, 179
get_frame_spec_data method, 177
get_frame_spec_data method, 175
get_frame_spec_data method, 173
get_frame_spec_data method, 171
get_frame_spec_data method, 169
get_frame_spec_data method, 167
get_frame_spec_data method, 164
get_frame_spec_data method, 162
get_frame_spec_data method, 160
get_frame_spec_data method, 158
get_frame_spec_data method, 156
get_frame_spec_data method, 154
get_frame_spec_data method, 152
get_frame_spec_data method, 150
get_frame_spec_data method, 148
get_frame_spec_data method, 146
get_frame_spec_data method, 144
get_frame_spec_data method, 142
get_frame_spec_data method, 140
get_frame_spec_data method, 138
get_frame_spec_data method, 136
get_frame_spec_data method, 134
get_frame_spec_data method, 132
get_frame_spec_data method, 130
get_frame_spec_data method, 128
get_frame_spec_data method, 126
get_frame_spec_data method, 124
get_frame_spec_data method, 122
get_frame_spec_data method, 120
get_frame_spec_data method, 118
get_frame_spec_data method, 116
get_frame_spec_data method, 114
get_frame_spec_data method, 112
get_frame_spec_data method, 110
get_frame_spec_data method, 108
get_frame_spec_data method, 106
get_frame_spec_data method, 104
get_frame_spec_data method, 102
get_frame_spec_data method, 100
get_frame_spec_data method, 98
get_frame_spec_data method, 96
get_frame_spec_data method, 94
get_frame_spec_data method, 92
get_frame_spec_data method, 90
get_frame_spec_data method, 88
get_frame_spec_data method, 86
get_frame_spec_data method, 84
get_frame_spec_data method, 82
get_frame_spec_data method, 80
get_frame_spec_data method, 78
get_frame_spec_data method, 76
get_frame_spec_data method, 74
get_frame_spec_data method, 72
get_frame_spec_data method, 70
get_frame_spec_data method, 68
get_frame_spec_data method, 66
get_frame_spec_data method, 64
get_frame_spec_data method, 62
get_frame_spec_data method, 60
get_frame_spec_data method, 58
get_frame_spec_data method, 56
get_frame_spec_data method, 54
get_frame_spec_data method, 52
get_frame_spec_data method, 50
get_frame_spec_data method, 48
get_frame_spec_data method, 46
get_frame_spec_data method, 44
get_frame_spec_data method, 42
get_frame_spec_data method, 40
get_frame_spec_data method, 38
get_frame_spec_data method, 36
get_frame_spec_data method, 34
get_frame_spec_data method, 32
get_frame_spec_data method, 30
get_frame_spec_data method, 28
get_frame_spec_data method, 26
get_frame_spec_data method, 24
get_frame_spec_data method, 22
get_frame_spec_data method, 20
get_frame_spec_data method, 18
get_frame_spec_data method, 16
get_frame_spec_data method, 14
get_frame_spec_data method, 12
get_frame_spec_data method, 10
get_frame_spec_data method, 8
get_frame_spec_data method, 6
get_frame_spec_data method, 4
get_frame_spec_data method, 2
get_frame_spec_data method, 0
get_frame_type() (digi.xbee.packets.raw.RX64IPacket method), 244
get_frame_type() (digi.xbee.packets.raw.RX64Packet get_frame_type() (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket method), 310
get_frame_type() (digi.xbee.packets.raw.TX16Packet get_frame_type() (digi.xbee.packets.zigbee.RouteRecordIndicatorPacket method), 238
get_frame_type() (digi.xbee.packets.raw.TX64Packet method), 233
get_frame_type() (digi.xbee.packets.raw.TXStatusPacket method), 230
get_frame_type() (digi.xbee.packets.relay.UserDataRelayOutputPacket method), 253
get_frame_type() (digi.xbee.packets.relay.UserDataRelayPacket method), 250
get_frame_type() (digi.xbee.packets.socket.SocketBindListenPacket method), 284
get_frame_type() (digi.xbee.packets.socket.SocketClosePacket method), 273
get_frame_type() (digi.xbee.packets.socket.SocketCloseResponsePacket method), 275
get_frame_type() (digi.xbee.packets.socket.SocketConnectPacket method), 267
get_frame_type() (digi.xbee.packets.socket.SocketConnectResponsePacket method), 270
get_frame_type() (digi.xbee.packets.socket.SocketCreatePacket method), 255
get_frame_type() (digi.xbee.packets.socket.SocketCreateResponsePacket method), 258
get_frame_type() (digi.xbee.packets.socket.SocketListenResponsePacket method), 287
get_frame_type() (digi.xbee.packets.socket.SocketNewIPv4ClientPacket method), 196
get_frame_type() (digi.xbee.packets.socket.SocketOptionRequestPacket method), 261
get_frame_type() (digi.xbee.packets.socket.SocketOptionResponsePacket method), 264
get_frame_type() (digi.xbee.packets.socket.SocketReceiveFromPacket method), 295
get_frame_type() (digi.xbee.packets.socket.SocketReceivePacket method), 292
get_frame_type() (digi.xbee.packets.socket.SocketSendPacket method), 278
get_frame_type() (digi.xbee.packets.socket.SocketSendToPacket method), 281
get_frame_type() (digi.xbee.packets.socket.SocketStatePacket method), 297
get_frame_type() (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 301
get_frame_type() (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket method), 307
get_frame_type() (digi.xbee.packets.wifi.RemoteATCommandWifiPacket method), 304
get_frame_type() (digi.xbee.packets.zigbee.RegisterDeviceStatusPacket method), 313
get_frame_type_value() (digi.xbee.packets.base.GenericXBeePacket method), 154
get_frame_type_value() (digi.xbee.packets.base.UnknownXBeePacket method), 156
get_frame_type_value() (digi.xbee.packets.base.XBeeAPIPacket method), 151
get_frame_type_value() (digi.xbee.packets.cellular.RXSMSPacket method), 159
get_frame_type_value() (digi.xbee.packets.cellular.TXSMSPacket method), 162
get_frame_type_value() (digi.xbee.packets.common.ATCommPacket method), 165
get_frame_type_value() (digi.xbee.packets.common.ATCommQueuePacket method), 196
get_frame_type_value() (digi.xbee.packets.common.ExplicitAddressingPacket method), 189
get_frame_type_value() (digi.xbee.packets.common.ModemStatusPacket method), 184
get_frame_type_value() (digi.xbee.packets.common.ReceivePacket method), 174
get_frame_type_value() (digi.xbee.packets.common.RemoteATCommandPacket method), 177
get_frame_type_value() (digi.xbee.packets.common.RemoteATCommandResponsePacket method), 180
get_frame_type_value() (digi.xbee.packets.common.TransmitPacket method), 184
get_frame_type_value() (digi.xbee.packets.common.TransmitStatusPacket method), 187
Index 815
get_frame_type_value()  (digi.xbee.packets.devicecloud.DeviceRequestPacket method), 187
get_frame_type_value()  (digi.xbee.packets.devicecloud.DeviceResponsePacket method), 203
get_frame_type_value()  (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket method), 206
get_frame_type_value()  (digi.xbee.packets.devicecloud.FrameErrorPacket method), 211
get_frame_type_value()  (digi.xbee.packets.devicecloud.SendDataRequestPacket method), 214
get_frame_type_value()  (digi.xbee.packets.devicecloud.SendDataResponsePacket method), 216
get_frame_type_value()  (digi.xbee.packets.devicecloud.RouteInformationPacket method), 220
get_frame_type_value()  (digi.xbee.packets.network.RXIPv4Packet method), 224
get_frame_type_value()  (digi.xbee.packets.network.TXIPv4Packet method), 227
get_frame_type_value()  (digi.xbee.packets.socket.SocketCreatePacket method), 255
get_frame_type_value()  (digi.xbee.packets.socket.SocketCreateResponsePacket method), 258
get_frame_type_value()  (digi.xbee.packets.socket.SocketOptionRequestPacket method), 261
get_frame_type_value()  (digi.xbee.packets.socket.SocketOptionResponsePacket method), 264
get_frame_type_value()  (digi.xbee.packets.socket.SocketSendPacket method), 278
get_frame_type_value()  (digi.xbee.packets.socket.SocketSendToPacket method), 281
get_frame_type_value()  (digi.xbee.packets.socket.SocketStatePacket method), 284
get_frame_type_value()  (digi.xbee.packets.socket.SocketBindListenPacket method), 287
get_frame_type_value()  (digi.xbee.packets.socket.SocketClosePacket method), 273
get_frame_type_value()  (digi.xbee.packets.socket.SocketCloseResponsePacket method), 276
get_frame_type_value()  (digi.xbee.packets.socket.SocketConnectPacket method), 267
get_frame_type_value()  (digi.xbee.packets.socket.SocketConnectResponsePacket method), 270
get_frame_type_value()  (digi.xbee.packets.socket.SocketListenResponsePacket method), 280
get_frame_type_value()  (digi.xbee.packets.socket.SocketNewIPv4ClientPacket method), 290
get_frame_type_value()  (digi.xbee.packets.socket.SocketReceiveFromPacket method), 293
get_frame_type_value()  (digi.xbee.packets.socket.SocketReceivePacket method), 295
get_frame_type_value()  (digi.xbee.packets.socket.SocketStatePacket method), 297
get_frame_type_value()  (digi.xbee.packets.socket.SocketStatePacket method), 299
get_frame_type_value()  (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 301
get_frame_type_value()  (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket method), 303

Index
get_frame_type_value()
(digi.xbee.packets.wifi.RemoteATCommandWifiPacket method), 304
get_frame_type_value()
(digi.xbee.packets.zigbee.RegisterDeviceStatusPacket method), 313
get_frame_type_value()
(digi.xbee.packets.zigbee.RegisterJoiningDevicePacket method), 311
get_frame_type_value()
(digi.xbee.packets.zigbee.RouteRecordIndicatorPacket method), 316
get_gateway_address()
(digi.xbee.devices.WiFiDevice method), 570
get_hardware_version()
(digi.xbee.devices.AbstractXBeeDevice method), 327
get_hardware_version()
(digi.xbee.devices.CellularDevice method), 504
get_hardware_version()
(digi.xbee.devices.DigiMeshDevice method), 398
get_hardware_version()
(digi.xbee.devices.DigiPointDevice method), 423
get_hardware_version()
(digi.xbee.devices.IPDevice method), 481
get_hardware_version()
(digi.xbee.devices.LPWANDevice method), 526
get_hardware_version()
(digi.xbee.devices.NBIoTDevice method), 550
get_hardware_version()
(digi.xbee.devices.RemoteDigiMeshDevice method), 630
get_hardware_version()
(digi.xbee.devices.RemoteDigiPointDevice method), 645
get_hardware_version()
(digi.xbee.devices.RemoteRaw802Device method), 615
get_hardware_version()
(digi.xbee.devices.RemoteXBeeDevice method), 601
get_hardware_version()
(digi.xbee.devices.RemoteZigBeeDevice method), 662
get_hardware_version()
(digi.xbee.devices.XBeeDevice method), 353
get_hardware_version()
(digi.xbee.devices.ZigBeeDevice method), 453
get_hsb() (digi.xbee.models.address.XBee16BitAddress method), 124
getimei_addr() (digi.xbee.devices.CellularDevice method), 494
getimei_addr() (digi.xbee.devices.NBIoTDevice method), 550
getimei_addr() (digi.xbee.devices.LPWANDevice method), 527
getint_from_byte() (in module digi.xbee.util.utils), 318
getio_configuration()
(digi.xbee.devices.AbstractXBeeDevice method), 330
getio_configuration()
(digi.xbee.devices.CellularDevice method), 504
getio_configuration()
(digi.xbee.devices.DigiMeshDevice method), 398
getio_configuration()
(digi.xbee.devices.DigiPointDevice method), 423
getio_configuration()
(digi.xbee.devices.IPDevice method), 481
getio_configuration()
(digi.xbee.devices.LPWANDevice method), 527
getio_configuration()
(digi.xbee.devices.NBIoTDevice method), 550
getio_configuration()
(digi.xbee.devices.RemoteDigiMeshDevice method), 630
getio_configuration()
(digi.xbee.devices.RemoteDigiPointDevice method), 645
getio_configuration()
(digi.xbee.devices.RemoteRaw802Device method), 615
getio_configuration()
(digi.xbee.devices.RemoteRaw802Device method), 615
getio_configuration()
(digi.xbee.devices.RemoteXBeeDevice method), 601
getio_configuration()
(digi.xbee.devices.RemoteZigBeeDevice method), 662
Index 817
get_io_configuration()
(digi.xbee.devices.ZigBeeDevice method), 353
get_io_configuration()
(digi.xbee.devices.XBeeDevice method), 354
get_io_configuration()
(digi.xbee.devices.CellularDevice method), 504
get_io_configuration()
(digi.xbee.devices.IPDevice method), 468
get_io_configuration()
(digi.xbee.devices.LPWANDevice method), 527
get_io_configuration()
(digi.xbee.devices.NBIoTDevice method), 551
get_io_configuration()
(digi.xbee.devices.WiFiDevice method), 580
get_io_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 454
get_ip_addr()
(digi.xbee.devices.ZigBeeDevice method), 454
get_ip_addr()
(digi.xbee.devices.CellularDevice method), 569
get_ip_addr()
(digi.xbee.devices.IPDevice method), 481
get_ip_addr()
(digi.xbee.devices.LPWANDevice method), 527
get_ip_addr()
(digi.xbee.devices.NBIoTDevice method), 551
get_ip_addr()
(digi.xbee.devices.WiFiDevice method), 581
get_ip_addressing_mode()
(digi.xbee.devices.WiFiDevice method), 569
get_ip_data_received_callbacks()
(digi.xbee.reader.PacketListener method), 767
get_ip_queue()
(digi.xbee.reader.PacketListener method), 762
get_local_xbee_device()
(digi.xbee.devices.RemoteDigiMeshDevice method), 631
get_local_xbee_device()
(digi.xbee.devices.RemoteDigiPointDevice method), 646
get_local_xbee_device()
(digi.xbee.devices.RemoteRaw802Device method), 616
get_local_xbee_device()
(digi.xbee.devices.RemoteXBeedevice method), 595
get_local_xbee_device()
(digi.xbee.devices.RemoteZigBeeDevice method), 662
get_lsb()
(digi.xbee.models.address.XBee16BitAddress method), 125
get_many_to_one_broadcasting_time()
(digi.xbee.devices.ZigBeeDevice method), 438
get_mask_address()
(digi.xbee.devices.WiFiDevice method), 570
get_micropython_data_received_callbacks()
(digi.xbee.reader.PacketListener method), 767
get_modem_status_received_callbacks()
(digi.xbee.reader.PacketListener method), 767
get_neighbors()
(digi.xbee.devices.DigiMeshDevice method), 331
get_neighbors()
(digi.xbee.devices.ZigBeeDevice method), 454
get_neighbors()
(digi.xbee.devices.CellularDevice method), 495
get_neighbors()
(digi.xbee.devices.DigiMeshDevice method), 398
get_neighbors()
(digi.xbee.devices.DigiPointDevice method), 423
get_neighbors()
(digi.xbee.devices.IPDevice method), 481
get_neighbors()
(digi.xbee.devices.LPWANDevice method), 527
get_neighbors()
(digi.xbee.devices.NBIoTDevice method), 551
get_neighbors()
(digi.xbee.devices.Raw802Device method), 495
get_neighbors()
(digi.xbee.devices.RemoteDigiMeshDevice method), 631
get_neighbors()
(digi.xbee.devices.RemoteDigiPointDevice method), 646
get_neighbors()
(digi.xbee.devices.RemoteRaw802Device method), 616
get_neighbors()
(digi.xbee.devices.RemoteXBeedevice method), 595
get_neighbors()
(digi.xbee.devices.RemoteZigBeeDevice method), 662
get_many_to_one_broadcasting_time()
(digi.xbee.devices.ZigBeeDevice method), 438
get_mask_address()
(digi.xbee.devices.WiFiDevice method), 570
get_micropython_data_received_callbacks()
(digi.xbee.reader.PacketListener method), 767
get_modem_status_received_callbacks()
(digi.xbee.reader.PacketListener method), 767
get_neighbors()
(digi.xbee.devices.DigiMeshDevice method), 389
get_neighbors()
(digi.xbee.devices.RemoteDigiMeshDevice method), 625
get_neighbors()
(digi.xbee.devices.RemoteZigBeeDevice method), 666
get_neighbors()
(digi.xbee.devices.ZigBeeDevice method), 444
get_network()
(digi.xbee.devices.CellularDevice method), 656
get_number_devices()
(digi.xbee.devices.ZigBeeNetwork method), 687

get_packet_received_callbacks()
(digi.xbee.reader.PacketListener method), 766

get_pan_id() (digi.xbee.devices.AbstractXBeeDevice method), 329
get_pan_id() (digi.xbee.devices.CellularDevice method), 505
get_pan_id() (digi.xbee.devices.DigiMeshDevice method), 399
get_pan_id() (digi.xbee.devices.DigiPointDevice method), 424
get_pan_id() (digi.xbee.devices.IPDevice method), 472
get_pan_id() (digi.xbee.devices.LPWANDevice method), 528
get_pan_id() (digi.xbee.devices.NBIoTDevice method), 551
get_pan_id() (digi.xbee.devices.Raw802Device method), 373
get_pan_id() (digi.xbee.devices.RemoteDigiMeshDevice method), 631
get_pan_id() (digi.xbee.devices.RemoteDigiPointDevice method), 646
get_pan_id() (digi.xbee.devices.RemoteRaw802Device method), 616
get_pan_id() (digi.xbee.devices.RemoteXBeeDevice method), 595
get_pan_id() (digi.xbee.devices.RemoteZigBeeDevice method), 663
get_pan_id() (digi.xbee.devices.WiFiDevice method), 581
get_pan_id() (digi.xbee.devices.XBeeDevice method), 340
get_pan_id() (digi.xbee.devices.ZigBeeDevice method), 455

going_power_level()
(digi.xbee.devices.AbstractXBeeDevice method), 329
going_power_level()
(digi.xbee.devices.CellularDevice method), 505
going_power_level()
(digi.xbee.devices.DigiMeshDevice method), 399
going_power_level()
(digi.xbee.devices.DigiPointDevice method), 424
going_power_level()
(digi.xbee.devices.IPDevice method), 482
going_power_level()
(digi.xbee.devices.LPWANDevice method), 528
going_power_level()
(digi.xbee.devices.NBIoTDevice method), 552
going_power_level()
(digi.xbee.devices.Raw802Device method), 373
going_power_level()
(digi.xbee.devices.RemoteDigiMeshDevice method), 632
going_power_level()
(digi.xbee.devices.RemoteDigiPointDevice method), 646

get_parameter() (digi.xbee.devices.AbstractXBeeDevice method), 631
get_parameter() (digi.xbee.devices.CellularDevice method), 496
get_parameter() (digi.xbee.devices.DigiMeshDevice method), 324
get_parameter() (digi.xbee.devices.DigiPointDevice method), 354
get_parameter() (digi.xbee.devices.IPDevice method), 454
get_parameter() (digi.xbee.devices.NBIoTDevice method), 481
get_parameter() (digi.xbee.devices.Raw802Device method), 482
get_parameter() (digi.xbee.devices.LPWANDevice method), 505
get_parameter() (digi.xbee.devices.ZigBeeDevice method), 455

going_power_level()
(digi.xbee.devices.AbstractXBeeDevice method), 329
-going_power_level()
(digi.xbee.devices.CellularDevice method), 505
-going_power_level()
(digi.xbee.devices.DigiMeshDevice method), 399
-going_power_level()
(digi.xbee.devices.DigiPointDevice method), 424
-going_power_level()
(digi.xbee.devices.IPDevice method), 482
-going_power_level()
(digi.xbee.devices.LPWANDevice method), 528
-going_power_level()
(digi.xbee.devices.NBIoTDevice method), 552
-going_power_level()
(digi.xbee.devices.Raw802Device method), 373
-going_power_level()
(digi.xbee.devices.RemoteDigiMeshDevice method), 632
-going_power_level()
(digi.xbee.devices.RemoteDigiPointDevice method), 646

get_pan_id() (digi.xbee.devices.ZigBeeNetwork method), 687
get_pan_id() (digi.xbee.devices.ZigBeeNetwork method), 687
XBee Python Library Documentation, Release 1.3.0

get_power_level() (digi.xbee.devices.RemoteRaw802Device method), 616
get_power_level() (digi.xbee.devices.RemoteXBeeDevice method), 602
get_power_level() (digi.xbee.devices.RemoteZigBeeDevice method), 663
get_power_level() (digi.xbee.devices.WiFiDevice method), 581
get_power_level() (digi.xbee.devices.XBeeDevice method), 455

get_power_level() (digi.xbee.devices.ZigBeeDevice method), 455
get_power_level() (digi.xbee.devices.ZigBeeDevice method), 438

get_protocol() (digi.xbee.devices.AbstractXBeeDevice method), 327
get_protocol() (digi.xbee.devices.CellularDevice method), 493
get_protocol() (digi.xbee.devices.DigiMeshDevice method), 387
get_protocol() (digi.xbee.devices.DigiPointDevice method), 413
get_protocol() (digi.xbee.devices.IPDevice method), 482
get_protocol() (digi.xbee.devices.LPWANDevice method), 541
get_protocol() (digi.xbee.devices.NBIoTDevice method), 528
get_protocol() (digi.xbee.devices.Raw802Device method), 363

get_protocol() (digi.xbee.devices.RemoteDigiMeshDevice method), 625
get_protocol() (digi.xbee.devices.RemoteDigiPointDevice method), 640
get_protocol() (digi.xbee.devices.RemoteRaw802Device method), 610
get_protocol() (digi.xbee.devices.Remote XBeeDevice method), 602
get_protocol() (digi.xbee.devices.RemoteZigBeeDevice method), 663

get_pwm_duty_cycle() (digi.xbee.devices.DigiMeshDevice method), 399
get_pwm_duty_cycle() (digi.xbee.devices.DigiMeshDevice method), 438
get_pwm_duty_cycle() (digi.xbee.devices.LPWANDevice method), 528
get_pwm_duty_cycle() (digi.xbee.devices.NBIoTDevice method), 552
get_pwm_duty_cycle() (digi.xbee.devices.Raw802Device method), 374

get_pwm_duty_cycle() (digi.xbee.devices.RemoteDigiMeshDevice method), 632
get_pwm_duty_cycle() (digi.xbee.devices.RemoteDigiPointDevice method), 647
get_pwm_duty_cycle() (digi.xbee.devices.RemoteRaw802Device method), 616
get_pwm_duty_cycle() (digi.xbee.devices.RemoteXBeeDevice method), 602
get_pwm_duty_cycle() (digi.xbee.devices.RemoteZigBeeDevice method), 663

get_pwm_duty_cycle() (digi.xbee.devices.WiFiDevice method), 582
get_pwm_duty_cycle() (digi.xbee.devices.XBeeDevice method), 355
get_pwm_duty_cycle() (digi.xbee.devices.ZigBeeDevice method), 455

get_queue() (digi.xbee.reader.PacketListener method), 762

get_read_timeout() (digi.xbee.serial.XBeeSerialPort method), 773

get_role() (digi.xbee.devices.AbstractXBeeDevice method), 328
get_role() (digi.xbee.devices.CellularDevice method), 505
get_role() (digi.xbee.devices.DigiMeshDevice method), 400
get_role() (digi.xbee.devices.DigiPointDevice method), 425
get_role() (digi.xbee.devices.IPDevice method), 482
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Class Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.LPWANDevice</code> method</td>
<td>529</td>
</tr>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.NBIoTDevice</code> method</td>
<td>552</td>
</tr>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.Raw802Device</code> method</td>
<td>374</td>
</tr>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.RemoteDigiMeshDevice</code> method</td>
<td>632</td>
</tr>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.RemoteDigiPointDevice</code> method</td>
<td>647</td>
</tr>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.RemoteRaw802Device</code> method</td>
<td>617</td>
</tr>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.RemoteXBeeDevice</code> method</td>
<td>602</td>
</tr>
<tr>
<td><code>get_role()</code></td>
<td><code>digi.xbee.devices.RemoteZigBeeDevice</code> method</td>
<td>663</td>
</tr>
<tr>
<td><code>get_route_info_callbacks()</code></td>
<td><code>digi.xbee.reader.PacketListener</code> method</td>
<td>768</td>
</tr>
<tr>
<td><code>get_route_record_received_callbacks()</code></td>
<td><code>digi.xbee.reader.PacketListener</code> method</td>
<td>768</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.CellularDevice</code> method</td>
<td>506</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.DigiMeshDevice</code> method</td>
<td>400</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.DigiPointDevice</code> method</td>
<td>425</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.IPDevice</code> method</td>
<td>483</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.LPWANDevice</code> method</td>
<td>529</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.NBIoTDevice</code> method</td>
<td>552</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.Raw802Device</code> method</td>
<td>374</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.WiFiDevice</code> method</td>
<td>582</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.XBeeDevice</code> method</td>
<td>348</td>
</tr>
<tr>
<td><code>get_route_to_node()</code></td>
<td><code>digi.xbee.devices.ZigBeeDevice</code> method</td>
<td>456</td>
</tr>
<tr>
<td><code>get_routes()</code></td>
<td><code>digi.xbee.devices.RemoteZigBeeDevice</code> method</td>
<td>656</td>
</tr>
<tr>
<td><code>get_routes()</code></td>
<td><code>digi.xbee.devices.ZigBeeDevice</code> method</td>
<td>444</td>
</tr>
<tr>
<td><code>get_serial_port()</code></td>
<td><code>digi.xbee.devices.RemoteDigiMeshDevice</code> method</td>
<td>632</td>
</tr>
<tr>
<td><code>get_serial_port()</code></td>
<td><code>digi.xbee.devices.RemoteDigiPointDevice</code> method</td>
<td>647</td>
</tr>
<tr>
<td><code>get_serial_port()</code></td>
<td><code>digi.xbee.devices.RemoteRaw802Device</code> method</td>
<td>617</td>
</tr>
<tr>
<td><code>get_serial_port()</code></td>
<td><code>digi.xbee.devices.RemoteXBeeDevice</code> method</td>
<td>596</td>
</tr>
<tr>
<td><code>get_serial_port()</code></td>
<td><code>digi.xbee.devices.RemoteZigBeeDevice</code> method</td>
<td>664</td>
</tr>
<tr>
<td><code>get_setting_default_value()</code></td>
<td><code>digi.xbee.profile.XBeeProfile</code> method</td>
<td>740</td>
</tr>
<tr>
<td><code>get_sms_received_callbacks()</code></td>
<td><code>digi.xbee.reader.PacketListener</code> method</td>
<td>767</td>
</tr>
<tr>
<td><code>get_sock_info()</code></td>
<td><code>digi.xbee.xsocket.socket</code> method</td>
<td>778</td>
</tr>
<tr>
<td><code>get_socket_data_received_callbacks()</code></td>
<td><code>digi.xbee.reader.PacketListener</code> method</td>
<td>768</td>
</tr>
<tr>
<td><code>get_socket_data_received_from_callbacks()</code></td>
<td><code>digi.xbee.reader.PacketListener</code> method</td>
<td>768</td>
</tr>
<tr>
<td><code>get_socket_info()</code></td>
<td><code>digi.xbee.devices.CellularDevice</code> method</td>
<td>495</td>
</tr>
<tr>
<td><code>get_socket_info()</code></td>
<td><code>digi.xbee.devices.LPWANDevice</code> method</td>
<td>529</td>
</tr>
<tr>
<td><code>get_socket_info()</code></td>
<td><code>digi.xbee.devices.NBIoTDevice</code> method</td>
<td>553</td>
</tr>
<tr>
<td><code>get_socket_state_received_callbacks()</code></td>
<td><code>digi.xbee.reader.PacketListener</code> method</td>
<td>767</td>
</tr>
<tr>
<td><code>get_sockets_list()</code></td>
<td><code>digi.xbee.devices.CellularDevice</code> method</td>
<td>494</td>
</tr>
<tr>
<td><code>get_sockets_list()</code></td>
<td><code>digi.xbee.devices.LPWANDevice</code> method</td>
<td>530</td>
</tr>
<tr>
<td><code>get_sockets_list()</code></td>
<td><code>digi.xbee.devices.NBIoTDevice</code> method</td>
<td>553</td>
</tr>
<tr>
<td><code>get_sync_ops_timeout()</code></td>
<td><code>digi.xbee.devices.AbstractXBeeDevice</code> method</td>
<td>328</td>
</tr>
</tbody>
</table>
get_sync_ops_timeout()
(digi.xbee.devices.CellularDevice method), 506
get_sync_ops_timeout()
(digi.xbee.devices.DigiMeshDevice method), 400
get_sync_ops_timeout()
(digi.xbee.devices.DigiPointDevice method), 425
get_sync_ops_timeout()
(digi.xbee.devices.IPDevice method), 483
get_sync_ops_timeout()
(digi.xbee.devices.LPWANDevice method), 530
get_sync_ops_timeout()
(digi.xbee.devices.NBIoTDevice method), 553
get_sync_ops_timeout()
(digi.xbee.devices.Raw802Device method), 375
get_sync_ops_timeout()
(digi.xbee.devices.WiFiDevice method), 583
get_sync_ops_timeout()
(digi.xbee.devices.XBeeDevice method), 347
get_xbee_device_callbacks()
(digi.xbee.devices.ZigBeeDevice method), 456
getblocking() (digi.xbee.xsocket.socket method), 775
getsocketopt() (digi.xbee.xsocket.socket method), 777
gettimeout() (digi.xbee.xsocket.socket method), 775

H

hardware_version (digi.xbee.profile.XBeeProfile attribute), 741
HardwareVersion (class in digi.xbee.models.hw), 119
has_analog_value() (digi.xbee.io.IOSample method), 733
has_analog_values() (digi.xbee.io.IOSample method), 733
has_devices() (digi.xbee.devices.DigiMeshNetwork method), 707
has_devices() (digi.xbee.devices.DigiPointNetwork method), 716
has_devices() (digi.xbee.devices.Raw802Network method), 697
has_devices() (digi.xbee.devices.XBeeNetwork method), 673
has_devices() (digi.xbee.devices.ZigBeeNetwork method), 688
has_digital_value() (digi.xbee.io.IOSample method), 733
has_digital_values() (digi.xbee.io.IOSample method), 733
has_explicit_packets()
Index

824

(digi.xbee.devices.CellularDevice method), 506

has_explicit_packets() (digi.xbee.devices.DigiMeshDevice method), 401

has_explicit_packets() (digi.xbee.devices.DigiPointDevice method), 426

has_explicit_packets() (digi.xbee.devices.IPDevice method), 483

has_explicit_packets() (digi.xbee.devices.LPWANDevice method), 530

has_explicit_packets() (digi.xbee.devices.NBIoTDevice method), 553

has_explicit_packets() (digi.xbee.devices.Raw802Device method), 375

has_explicit_packets() (digi.xbee.devices.ZigBeeDevice method), 456

has_filesystem (digi.xbee.profile.XBeeProfile attribute), 741

has_packets() (digi.xbee.devices.CellularDevice method), 506

has_packets() (digi.xbee.devices.DigiMeshDevice method), 401

has_packets() (digi.xbee.devices.DigiPointDevice method), 426

has_packets() (digi.xbee.devices.IPDevice method), 483

has_packets() (digi.xbee.devices.LPWANDevice method), 530

has_packets() (digi.xbee.devices.NBIoTDevice method), 554

has_packets() (digi.xbee.devices.Raw802Device method), 375

has_packets() (digi.xbee.devices.WiFiDevice method), 583

has_packets() (digi.xbee.devices.XBeeDevice method), 344

has_power_supply_value() (digi.xbee.io.IOSample method), 733

has_pwm_capability() (digi.xbee.io.IOLine method), 731

hex_string_to_bytes() (in module digi.xbee.util.utils), 319

hex_to_string() (in module digi.xbee.util.utils), 321

hops (digi.xbee.packets.zigbee.RouteRecordIndicatorPacket attribute), 318

I2C_FUNCTIONALITY (digi.xbee.io.IOMode attribute), 735

id (digi.xbee.models.protocol.Role attribute), 137

ident (digi.xbee.reader.PacketListener attribute), 768

index (digi.xbee.io.IOLine attribute), 731

index (digi.xbee.profile.FirmwareBaudrate attribute), 735

index (digi.xbee.profile.FirmwareParity attribute), 736

index (digi.xbee.profile.FirmwareStopbits attribute), 736

index() (digi.xbee.reader.BluetoothDataReceived method), 753

index() (digi.xbee.reader.DataReceived method), 744

index() (digi.xbee.reader.DeviceDiscovered method), 747

index() (digi.xbee.reader.DiscoveryProcessFinished method), 748

index() (digi.xbee.reader.EndDiscoveryScan method), 760

index() (digi.xbee.reader.ExplicitDataReceived method), 749

index() (digi.xbee.reader.InitDiscoveryScan method), 760

index() (digi.xbee.reader.IOSampleReceived method), 746

index() (digi.xbee.reader.IPDataReceived method), 750

index() (digi.xbee.reader.MicroPythonDataReceived method), 753

index() (digi.xbee.reader.ModemStatusReceived method), 745

index() (digi.xbee.reader.NetworkModified method), 747

index() (digi.xbee.reader.PacketReceived method), 743

index() (digi.xbee.reader.RelayDataReceived method), 752

index() (digi.xbee.reader.RouteInformationReceived method), 758

index() (digi.xbee.reader.RouteReceived method), 759

index() (digi.xbee.reader.RouteRecordIndicatorReceived method), 757

index() (digi.xbee.reader.SMSSent method), 751

index() (digi.xbee.reader.SocketDataReceived method), 755

index() (digi.xbee.reader.SocketDataReceivedFrom method), 756
index() (digi.xbee.reader.SocketStateReceived method), 754
index() (digi.xbee.reader.XBeeEvent method), 742
InitDiscoveryScan (class in digi.xbee.reader), 759
insert() (digi.xbee.reader.RelayDataReceived method), 752
insert() (digi.xbee.reader.SocketStateReceived method), 754
insert() (digi.xbee.reader.XBeeEvent method), 742
int_to_ascii() (in module digi.xbee.util.utils), 320
int_to_bytes() (in module digi.xbee.util.utils), 319
int_to_length() (in module digi.xbee.util.utils), 321
InvalidConfigurationException, 724
InvalidOperatingModeException, 724
InvalidPacketException, 724
io_sample (digi.xbee.packets.raw.RX16IOPacket attribute), 192
io_sample (digi.xbee.packets.raw.RX16IOPacket attribute), 248
io_sample (digi.xbee.packets.raw.RX64IOPacket attribute), 244
io_sample (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket attribute), 301
IODataSampleRxIndicatorPacket (class in digi.xbee.packets.common), 190
IODataSampleRxIndicatorWifiPacket (class in digi.xbee.packets.wifi), 299
IOLine (class in digi.xbee.io), 730
IPMode (class in digi.xbee.io), 734
IOPacketReceived (class in digi.xbee.reader), 745
IPValue (class in digi.xbee.io), 731
is_apply_changes_enabled() (digi.xbee.devices.AbstractXBeeDevice method), 328
is_apply_changes_enabled() (digi.xbee.devices.CellularDevice method), 507
is_apply_changes_enabled() (digi.xbee.devices.DigiMeshDevice method), 401
is_apply_changes_enabled() (digi.xbee.devices.DigiPointDevice method), 426
is_apply_changes_enabled() (digi.xbee.devices.IPDevice method), 484
is_apply_changes_enabled() (digi.xbee.devices.LPWANDevice method), 530
is_apply_changes_enabled() (digi.xbee.devices.NBIoTDevice method), 554
is_apply_changes_enabled() (digi.xbee.devices.Raw802Device method), 375
io_sample (digi.xbee.packets.common.IODataSampleRxIndicatorPacket attribute), 192
is_apply_changes_enabled() (digi.xbee.devices.RemoteDigiMeshDevice method), 633
is_apply_changes_enabled() (digi.xbee.devices.RemoteDigiPointDevice method), 647
is_apply_changes_enabled() (digi.xbee.devices.RemoteRaw802Device method), 617
is_apply_changes_enabled() (digi.xbee.devices.RemoteXBeeDevice method), 603
is_apply_changes_enabled() (digi.xbee.devices.ZigBeeDevice method), 664
is_apply_changes_enabled() (digi.xbee.devices.WiFiDevice method), 583
is_apply_changes_enabled() (digi.xbee.devices.XBeeDevice method), 355
is_apply_changes_enabled() (digi.xbee.devices.ZigBeeDevice method), 457
is_bit_enabled() (in module digi.xbee.util.utils), 318
is_broadcast (digi.xbee.models.message.ExplicitXBeeMessage broadcast) (digi.xbee.models.message.TXIPv4Packet broadcast attribute), 128
is_broadcast (digi.xbee.models.message.XBeeMessage broadcast) (digi.xbee.models.raw.RX16IOPacket broadcast attribute), 127
is_broadcast () (digi.xbee.packets.base.GenericXBeePacket broadcast) (digi.xbee.packets.base.UnknownXBeePacket broadcast) (digi.xbee.packets.raw.RX64IOPacket broadcast) (digi.xbee.packets.raw.RX16Packet broadcast method), 154
is_broadcast () (digi.xbee.packets.base.UnknownXBeePacket broadcast) (digi.xbee.packets.raw.RX64IOPacket broadcast method), 156
is_broadcast () (digi.xbee.packets.base.XBeeAPacket broadcast) (digi.xbee.packets.raw.RX64Packet broadcast method), 151
is_broadcast () (digi.xbee.packets.cellular.RXSMSPacket broadcast) (digi.xbee.packets.cellular.TXSMSPacket broadcast) (digi.xbee.packets.raw.TX16Packet broadcast method), 159
is_broadcast () (digi.xbee.packets.cellular.TXSMSPacket broadcast) (digi.xbee.packets.raw.TX64Packet broadcast method), 162
is_broadcast () (digi.xbee.packets.common.ATCommPacket broadcast) (digi.xbee.packets.common.ATCommResponsePacket broadcast) (digi.xbee.packets.common.ATCommQueuePacket broadcast) (digi.xbee.packets.common.TreeStatusPacket broadcast method), 165
is_broadcast () (digi.xbee.packets.common.ATCommResponsePacket broadcast) (digi.xbee.packets.relay.UserDataRelayOutputPacket broadcast method), 171
is_broadcast () (digi.xbee.packets.common.ExplicitRXIndicatorPacket broadcast) (digi.xbee.packets.socket.SocketClosePacket broadcast method), 200
is_broadcast () (digi.xbee.packets.common.IODataSampleRxIndicatorPacket broadcast) (digi.xbee.packets.socket.SocketCloseResponsePacket broadcast) (digi.xbee.packets.socket.SocketConnectPacket broadcast method), 191
is_broadcast () (digi.xbee.packets.common.ModemStatusPacket broadcast) (digi.xbee.packets.socket.SocketConnectPacket broadcast method), 189
is_broadcast() (digi.xbee.packets.socket.SocketCreatePacket, 256)
is_broadcast() (digi.xbee.packets.socket.SocketCreateResponsePacket, 259)
is_broadcast() (digi.xbee.packets.socket.SocketListenResponsePacket, 287)
is_broadcast() (digi.xbee.packets.socket.SocketNewIPv4ClientPacket, 290)
is_broadcast() (digi.xbee.packets.socket.SocketOptionRequestPacket, 261)
is_broadcast() (digi.xbee.packets.socket.SocketOptionResponsePacket, 264)
is_broadcast() (digi.xbee.packets.socket.SocketReceiveFromPacket, 295)
is_broadcast() (digi.xbee.packets.socket.SocketReceivePacket, 293)
is_broadcast() (digi.xbee.packets.socket.SocketSendPacket, 279)
is_broadcast() (digi.xbee.packets.socket.SocketSendToPacket, 282)
is_broadcast() (digi.xbee.packets.socket.SocketStatePacket, 298)
is_broadcast() (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket, 302)
is_broadcast() (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket, 308)
is_broadcast() (digi.xbee.packets.wifi.RemoteATCommandWifiPacket, 305)
is_broadcast() (digi.xbee.packets.zigbee.RegisterDeviceStatusPacket, 313)
is_broadcast() (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket, 311)
is_broadcast() (digi.xbee.packets.zigbee.RouteRecordIndicatorPacket, 315)
is_connected() (digi.xbee.filesystem.LocalXBeeFileSystemManager, 726)
is_connected() (digi.xbee.xsocket.socket, 778)
is_connected() (digi.xbee.devices.CellularDevice, 493)
is_connected() (digi.xbee.devices.LPWANDevice, 531)
is_connected() (digi.xbee.devices.NBIoTDevice, 554)
is_connected() (digi.xbee.devices.WiFiDevice, 583)
is_connected() (digi.xbee.devices.XBeeDevice, 347)
is_discovery_running() (digi.xbee.devices.DigiMeshNetwork, 707)
is_discovery_running() (digi.xbee.devices.DigiPointNetwork, 716)
is_discovery_running() (digi.xbee.devices.XBeeNetwork, 673)
is_discovery_running() (digi.xbee.devices.ZigBeeNetwork, 688)
is_interface_open (digi.xbee.comm_interface.XBeeCommunicationInterface, 323)
is_open() (digi.xbee.devices.CellularDevice, 507)
is_open() (digi.xbee.devices.DigiMeshDevice, 401)
is_open() (digi.xbee.devices.DigiPointDevice, 426)
is_open() (digi.xbee.devices.IPDevice, 484)
is_open() (digi.xbee.devices.LPWANDevice, 531)
is_open() (digi.xbee.devices.NBIoTDevice, 554)
is_open() (digi.xbee.devices.Raw802Device, 375)
is_open() (digi.xbee.devices.RemoteDigiMeshDevice, 633)
is_open() (digi.xbee.devices.RemoteDigiPointDevice, 648)
is_remote() (digi.xbee.devices.CellularDevice, 507)
is_remote() (digi.xbee.devices.DigiMeshDevice, 401)
is_remote() (digi.xbee.devices.DigiPointDevice, 426)
is_remote() (digi.xbee.devices.IPDevice, 484)
is_remote() (digi.xbee.devices.LPWANDevice, 531)
is_remote() (digi.xbee.devices.NBIoTDevice, 554)
is_remote() (digi.xbee.devices.Raw802Device, 375)
is_remote() (digi.xbee.devices.RemoteDigiMeshDevice, 633)
is_remote() (digi.xbee.devices.RemoteDigiPointDevice, 648)
is_directory (digi.xbee.filesystem.FileSystemElement, 726)
is_discovery_running() (digi.xbee.devices.DigiMeshNetwork, 707)
is_discovery_running() (digi.xbee.devices.DigiPointNetwork, 716)
is_remote() (digi.xbee.devices.RemoteRaw802Device method), 617
is_remote() (digi.xbee.devices.RemoteXBeeDevice method), 595
is_remote() (digi.xbee.devices.RemoteZigBeeDevice method), 664
is_remote() (digi.xbee.devices.WiFiDevice method), 584
is_remote() (digi.xbee.devices.XBeeDevice method), 347
is_remote() (digi.xbee.devices.ZigBeeDevice method), 457
is_rssi (digi.xbee.devices.LinkQuality attribute), 720
is_valid() (digi.xbee.models.address.XBee16BitAddress attribute), 124
is_valid() (digi.xbee.models.address.XBee64BitAddress attribute), 125
is_valid() (digi.xbee.models.address.XBeeIMEIAddress attribute), 126
isAlive() (digi.xbee.reader.PacketListener method), 768
join() (digi.xbee.reader.PacketListener method), 768
join() (digi.xbee.reader.XBeeQueue method), 771
key (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket attribute), 310
length (digi.xbee.packets.digimesh.RouteInformationPacket attribute), 219
length_to_int() (in module digi.xbee.util.utils), 320
LinkQuality (class in digi.xbee.devices), 720
list_directory() (digi.xbee.filesystem.LocalXBeeFileSystemManager method), 727
listen() (digi.xbee.xsocket.socket method), 775
local_interface (digi.xbee.models.message.SendMessageData attribute), 130
LocalXBeeFileSystemManager (class in digi.xbee.filesystem), 726
log (digi.xbee.devices.AbstractXBeeDevice attribute), 338
log (digi.xbee.devices.CellularDevice attribute), 307
log (digi.xbee.devices.DigiMeshDevice attribute), 401
log (digi.xbee.devices.DigiPointDevice attribute), 426
log (digi.xbee.devices.IPDevice attribute), 484
log (digi.xbee.devices.LPWANDevice attribute), 531
log (digi.xbee.devices.NBIoTDevice attribute), 554
log (digi.xbee.devices.Raw802Device attribute), 376
log (digi.xbee.devices.RemoteDigiMeshDevice attribute), 633
log (digi.xbee.devices.RemoteDigiPointDevice attribute), 648
log (digi.xbee.devices.RemoteRaw802Device attribute), 618
log (digi.xbee.devices.RemoteXBeeDevice attribute), 603
log (digi.xbee.devices.RemoteZigBeeDevice attribute), 664
log (digi.xbee.devices.WiFiDevice attribute), 584
log (digi.xbee.devices.XBeeDevice attribute), 355
log (digi.xbee.devices.ZigBeeDevice attribute), 457
LOG_PATTERN (digi.xbee.devices.AbstractXBeeDevice attribute), 324
make_directory() (digi.xbee.filesystem.LocalXBeeFileSystemManager method), 727
MAX_TIME_BETWEEN_REQUESTS (digi.xbee.devices.XBeeNetwork attribute), 672
MAX_TIME_BETWEEN_REQUESTS (digi.xbee.devices.XBeeNetwork attribute), 671
MIN_TIME_BETWEEN_REQUESTS (digi.xbee.devices.XBeeNetwork attribute), 672
MIN_TIME_BETWEEN_REQUESTS (digi.xbee.devices.XBeeNetwork attribute), 671
modem_status (digi.xbee.packets.common.ModemStatusPacket attribute), 188
ModemStatusPacket (class in digi.xbee.models.status), 139
move_element() (digi.xbee.filesystem.LocalXBeeFileSystemManager method), 728
name (digi.xbee.filesystem.FileSystemElement attribute), 726
Index
NetworkDiscoveryStatus (class in digi.xbee.models.status), 144
NetworkEventReason (class in digi.xbee.devices), 720
NetworkEventType (class in digi.xbee.devices), 719
NetworkModified (class in digi.xbee.reader), 746
node_a (digi.xbee.devices.Connection attribute), 721
node_b (digi.xbee.devices.Connection attribute), 721
NONE (digi.xbee.models.options.ReceiveOptions attribute), 130
NONE (digi.xbee.models.options.RemoteATCmdOptions attribute), 132
NONE (digi.xbee.models.options.TransmitOptions attribute), 131
number_of_hops (digi.xbee.packets.zigbee.RouteRecordIndicatorPacket attribute), 317

open () (digi.xbee.comm_interface.XBeeCommunicationInterface method), 322
open () (digi.xbee.devices.CellularDevice method), 492
digi.xbee.devices.DigiMeshDevice method), 387
open () (digi.xbee.devices.DigiPointDevice method), 413
open () (digi.xbee.devices.IPDevice method), 484
open () (digi.xbee.devices.LPWANDevice method), 531
digi.xbee.devices.NBLoTDevice method), 541
open () (digi.xbee.devices.Raw802Device method), 362
digi.xbee.devices.WiFiDevice method), 565
digi.xbee.devices.XBeDevice method), 339
digi.xbee.devices.ZigBeeDevice method), 438
digi.xbee.devices.ZigBeeSerialPort method), 772
open () (digi.xbee.devices.DigiDevice attribute), 401
open () (digi.xbee.devices.DigiMeshDevice attribute), 507
open () (digi.xbee.devices.DigiMeshDevice attribute), 401
open () (digi.xbee.packets.socket.SocketStatePacket operating_mode (digi.xbee.devices.DigiPointDevice attribute), 299
needs_id() (digi.xbee.packets.socket.SocketSampleRxIndicatorPacket operating_mode (digi.xbee.devices.IPDevice attribute), 426
needs_id() (digi.xbee.packets.wifi.ODataSampleRxIndicatorPacket operating_mode (digi.xbee.devices.LPWANDevice attribute), 484
needs_id() (digi.xbee.packets.wifi.RemoteATCommandResponsePacket operating_mode (digi.xbee.devices.NBLoTDevice attribute), 531
needs_id() (digi.xbee.packets.wifi.RemoteATCommandResponsePacket operating_mode (digi.xbee.devices.Raw802Device attribute), 555
needs_id() (digi.xbee.packets.zigbee.RegisterDeviceStatusPacket operating_mode (digi.xbee.devices.WiFiDevice attribute), 376
needs_id() (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket operating_mode (digi.xbee.devices.XBeDevice attribute), 384
needs_id() (digi.xbee.packets.zigbee.RouteRecordIndicatorPacket operating_mode (digi.xbee.devices.XBeDevice attribute), 349
NeighborDiscoveryMode (class in digi.xbee.models.mode), 122
operating_mode (digi.xbee.devices.ZigBeeDevice attribute), 457
OperatingMode (class in digi.xbee.models.mode), 121
OperationNotSupportedException, 724
option (digi.xbee.packets.socket.SocketOptionRequestPacket attribute), 260
option (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
option_data (digi.xbee.packets.socket.SocketOptionRequestPacket attribute), 260
option_data (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
options (digi.xbee.packets.devicecloud.SendDataRequestPacket attribute), 213
options (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket attribute), 310
DigiPointDevice
SE_SOCKET
digi.xbee.packets.network.TXIPv4Packet attribute), 226
OPTIONS_LEAVE_SOCKET_OPEN
digi.xbee.packets.network.TXIPv4Packet attribute), 226
output () (digi.xbee.packets.base.GenericXBePacket method), 154
output () (digi.xbee.packets.base.UnknownXBePacket method), 157
output () (digi.xbee.packets.base.XBeAPIPacket method), 152
output () (digi.xbee.packets.base.XBePacket method), 149
output () (digi.xbee.packets.cellular.RXSMSPacket attribute), 159
output () (digi.xbee.packets.cellular.TXSMSPacket method), 162
output () (digi.xbee.packets.common.ATCommPacket method), 165
output () (digi.xbee.packets.common.ATCommQueuePacket method), 168
output () (digi.xbee.packets.common.ATCommResponsePacket method), 171
output() (digi.xbee.packets.common.ExplicitAddressingPacket method), 197
output() (digi.xbee.packets.common.ExplicitRXIndicatorPacket method), 199
output() (digi.xbee.packets.common.IODataSampleRxIndicatorPacket method), 193
output() (digi.xbee.packets.common.ModemStatusPacket method), 190
output() (digi.xbee.packets.common.ReceivePacket method), 174
output() (digi.xbee.packets.common.RemoteATCommandPacket method), 177
output() (digi.xbee.packets.common.RemoteATCommandResponsePacket method), 180
output() (digi.xbee.packets.common.TransmitPacket method), 184
output() (digi.xbee.packets.common.TransmitStatusPacket method), 187
output() (digi.xbee.packets.devicecloud.DeviceRequestPacket method), 203
output() (digi.xbee.packets.devicecloud.DeviceResponsePacket method), 206
output() (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket method), 209
output() (digi.xbee.packets.devicecloud.FrameErrorPacket method), 211
output() (digi.xbee.packets.devicecloud.SendDataRequestPacket method), 214
output() (digi.xbee.packets.devicecloud.SendDataResponsePacket method), 216
output() (digi.xbee.packets.digimesh.RouteInformationPacket method), 221
output() (digi.xbee.packets.networt.RXIPv4Packet method), 224
output() (digi.xbee.packets.networt.TXIPv4Packet method), 227
output() (digi.xbee.packets.raw.RX16IOPacket method), 247
output() (digi.xbee.packets.raw.RX16Packet method), 242
output() (digi.xbee.packets.raw.RX64IOPacket method), 245
output() (digi.xbee.packets.raw.RX64Packet method), 239
output() (digi.xbee.packets.raw.TX16Packet method), 233
output() (digi.xbee.packets.raw.TX64Packet method), 230
output() (digi.xbee.packets.raw.TXStatusPacket method), 236
output() (digi.xbee.packets.relay.UserDataRelayPacket method), 253
output() (digi.xbee.packets.relay.UserDataRelayOutputPacket method), 250

The XBee Python Library Documentation, Release 1.3.0

Index 831

PACKET_ACKNOWLEDGED
(digi.xbee.models.options.ReceiveOptions attribute), 130
PacketListener (class in digi.xbee.reader), 761
getReceived (class in digi.xbee.reader), 742
parameter (digi.xbee.models.atcomm.ATCommand attribute), 118
parameter (digixbee.packets.common.ATCommPacket attribute), 164
parameter (digixbee.packets.common.ATCommQueuePacket attribute), 167
parameter (digixbee.packets.common.RemoteATCommandPacket attribute), 176
parameter (digixbee.packets.wifi.RemoteATCommandWifiPacket attribute), 304
parent (digixbe.devices.RemoteZigBeeDevice attribute), 655
parity (digixbee.profile.FirmwareParity attribute), 736
path (digixbee.filesystem.FileSystemElement attribute), 726
path (digixbee.packets.devicecloud.SendDataRequestPacket attribute), 213
PATTERN (digixbee.models.address.XBee16BitAddress attribute), 124
PATTERN (digixbee.models.address.XBee64BitAddress attribute), 125
PATTERN (digixbee.models.address.XBeeIMEIAddress attribute), 126
PATTERN_PHONE_NUMBER (in module digixbee.packets.cellular), 157
payload (digixbee.packets.socket.SocketReceiveFromPacket attribute), 296
payload (digixbee.packets.socket.SocketReceivePacket attribute), 292
payload (digixbee.packets.socket.SocketSendPacket attribute), 278
payload (digixbee.packets.socket.SocketSendToPacket attribute), 281
phone_number (digixbee.models.message.SMSMessage attribute), 129
phone_number (digixbee.packets.cellular.RXSMSPacket attribute), 158
phone_number (digixbee.packets.cellular.TXSMSPacket attribute), 163
POINT_MULTIPOINT_MODE (digixbee.models.options.TransmitOptions attribute), 132
pop () (digixbee.reader.IPDataReceived method), 750
pop () (digixbee.reader.MicroPythonDataReceived method), 733
pop () (digixbee.reader.ModemStatusReceived method), 745
pop () (digixbee.reader.NetworkModified method), 747
pop () (digixbee.reader.PacketReceived method), 743
pop () (digixbee.reader.RelayDataReceived method), 752
pop () (digixbee.reader.RSMReceived method), 751
pop () (digixbee.reader.SocketDataReceived method), 755
pop () (digixbee.reader.SocketDataReceivedFrom method), 756
pop () (digixbee.reader.SocketStateReceived method), 754
pop () (digixbee.reader.XBeeEvent method), 742
power_supply_value (digixbee.io.IOSample attribute), 734
profile_id (digixbee.models.message.ExplicitXBeeMessage attribute), 128
profile_id (digixbee.packets.common.ExplicitAddressingPacket attribute), 197
profile_id (digixbee.packets.common.ExplicitRXIndicatorPacket attribute), 201
profile_settings (digixbee.models.XBeeProfile attribute), 741
protocol (digixbee.models.message.IPMessage attribute), 129
protocol (digixbee.packets.socket.SocketCreatePacket attribute), 255
purge_port () (digixbee.serial.XBeeSerialPort method), 773
put () (digixbee.reader.XBeeQueue method), 771
put_dir () (digixbee.filesystem.LocalXBeeFileSystemManager method), 728
put_file () (digixbee.filesystem.LocalXBeeFileSystemManager method), 728
put_nowait () (digixbee.reader.XBeeQueue method), 771
PWM (digixbee.io.IOMode attribute), 734
pwm_at_command (digixbee.io.IOLine attribute), 731
Q
qsize () (digixbee.reader.XBeeQueue method), 771
quit_reading() (digi.xbee.comm_interface.XBeeCommunicationInterface method), 323
quit_reading() (digi.xbee.serial.XBeeSerialPort method), 773

R
Raw802Device (class in digi.xbee.devices), 362
Raw802Network (class in digi.xbee.devices), 690
reachable (digi.xbee.devices.AbstractXBeeDevice attribute), 338
reachable (digi.xbee.devices.CellularDevice attribute), 507
reachable (digi.xbee.devices.DigiMeshDevice attribute), 402
reachable (digi.xbee.devices.DigiPointDevice attribute), 427
reachable (digi.xbee.devices.IPDevice attribute), 484
reachable (digi.xbee.devices.LPWANDevice attribute), 531
reachable (digi.xbee.devices.NBIoTDevice attribute), 555
reachable (digi.xbee.devices.Raw802Device attribute), 376
reachable (digi.xbee.devices.RemoteDigiMeshDevice attribute), 633
reachable (digi.xbee.devices.RemoteDigiPointDevice attribute), 648
reachable (digi.xbee.devices.RemoteRaw802Device attribute), 618
reachable (digi.xbee.devices.RemoteXBeeDevice attribute), 603
reachable (digi.xbee.devices.RemoteZigBeeDevice attribute), 664
reachable (digi.xbee.devices.WiFiDevice attribute), 584
reachable (digi.xbee.devices.XBeeDevice attribute), 355
reachable (digi.xbee.devices.ZigBeeDevice attribute), 457
read_byte() (digi.xbee.serial.XBeeSerialPort method), 773
read_bytes() (digi.xbee.serial.XBeeSerialPort method), 773
read_data() (digi.xbee.devices.CellularDevice method), 507
read_data() (digi.xbee.devices.DigiMeshDevice method), 402
read_data() (digi.xbee.devices.DigiPointDevice method), 427
read_data() (digi.xbee.devices.IPDevice method), 473
read_data() (digi.xbee.devices.ZigBeeDevice method), 457
read_data_from() (digi.xbee.devices.CellularDevice method), 507
read_data_from() (digi.xbee.devices.DigiMeshDevice method), 402
read_data_from() (digi.xbee.devices.DigiPointDevice method), 402
read_data_from() (digi.xbee.devices.IPDevice method), 473
read_data_from() (digi.xbee.devices.LPWANDevice method), 532
read_data_from() (digi.xbee.devices.NBIoTDevice method), 555
read_data_from() (digi.xbee.devices.Raw802Device method), 376
read_data_from() (digi.xbee.devices.WiFiDevice method), 584
read_data_from() (digi.xbee.devices.XBeeDevice method), 343
read_data_from() (digi.xbee.devices.ZigBeeDevice method), 458
read_device_info() (digi.xbee.devices.AbstractXBeeDevice method), 326
read_device_info() (digi.xbee.devices.CellularDevice method), 493
read_device_info() (digi.xbee.devices.DigiMeshDevice method), 403
read_device_info() (digi.xbee.devices.DigiPointDevice method), 428
read_device_info() (digi.xbee.devices.IPDevice method), 468
read_device_info() (digi.xbee.devices.LPWANDevice method), 532
read_device_info() (digi.xbee.devices.NBIoTDevice method), 555
read_device_info() (digi.xbee.devices.Raw802Device method), 377
read_device_info()
read_device_info()
(digi.xbee.devices.RemoteDigiMeshDevice method), 633
read_device_info()
(digi.xbee.devices.RemoteDigiPointDevice method), 648
read_device_info()
(digi.xbee.devices.RemoteRaw802Device method), 618
read_device_info()
(digi.xbee.devices.RemoteXBeeDevice method), 603
read_device_info()
(digi.xbee.devices.RemoteZigBeeDevice method), 665
read_device_info()
(digi.xbee.devices.WiFiDevice method), 584
read_device_info()
(digi.xbee.devices.XBeeDevice method), 356
read_device_info()
(digi.xbee.devices.ZigBeeDevice method), 459
read_expl_data()
(digi.xbee.devices.CellularDevice method), 508
read_expl_data()
(digi.xbee.devices.DigiMeshDevice method), 388
read_expl_data()
(digi.xbee.devices.DigiPointDevice method), 414
read_expl_data()
(digi.xbee.devices.ZigBeeDevice method), 439
read_expl_data_from()
(digi.xbee.devices.CellularDevice method), 508
read_expl_data_from()
(digi.xbee.devices.DigiMeshDevice method), 388
read_expl_data_from()
(digi.xbee.devices.DigiPointDevice method), 414
read_expl_data_from()
(digi.xbee.devices.ZigBeeDevice method), 439
read_io_sample()
(digi.xbee.devices.AbstractXBeeDevice method), 331
read_io_sample()
(digi.xbee.devices.CellularDevice method), 507
read_io_sample()
(digi.xbee.devices.DigiMeshDevice method), 403
read_io_sample()
(digi.xbee.devices.DigiPointDevice method), 428
read_io_sample()
(digi.xbee.devices.IPDevice method), 471
read_io_sample()
(digi.xbee.devices.LPWANDevice method), 533
read_io_sample()
(digi.xbee.devices.NBIoTDevice method), 556
read_io_sample()
(digi.xbee.devices.Raw802Device method), 485
read_io_sample()
(digi.xbee.devices.LPWANDevice method), 532
read_io_sample()
(digi.xbee.devices.NBIoTDevice method), 555
read_io_sample()
(digi.xbee.devices.WiFiDevice method), 585
read_profile_exception
(digi.xbee.devices.WiFiDevice method), 739
real_status
(digi.xbee.packets.common.ATCommResponsePacket attribute), 170
real_status
(digi.xbee.packets.common.RemoteATCommandResponsePacket attribute), 179
receive_options
(digi.xbee.packets.common.ExplicitRXIndicatorPacket attribute), 200
receive_options
(digi.xbee.packets.common.IODataSampleRxIndicatorPacket attribute), 192
receive_options
(digi.xbee.packets.raw.RX16IOPacket attribute), 248
receive_options
(digi.xbee.packets.raw.RX16Packet attribute), 248
receive_options
(digi.xbee.packets.raw.RX16Packet method), 377
receive_options
(digi.xbee.devices.RemoteDigiMeshDevice method), 634
receive_options
(digi.xbee.devices.RemoteDigiPointDevice method), 648
receive_options
(digi.xbee.devices.RemoteRaw802Device method), 618
receive_options
(digi.xbee.devices.RemoteXBeeDevice method), 603
receive_options
(digi.xbee.devices.RemoteZigBeeDevice method), 665
receive_options
(digi.xbee.devices.WiFiDevice method), 584
receive_options
(digi.xbee.devices.XBeeDevice method), 356
receive_options
(digi.xbee.devices.ZigBeeDevice method), 459
read_ip_data()
(digi.xbee.devices.CellularDevice method), 508
read_ip_data()
(digi.xbee.devices.DigiMeshDevice method), 388
read_ip_data()
(digi.xbee.devices.DigiPointDevice method), 414
read_ip_data()
(digi.xbee.devices.IPDevice method), 471
read_ip_data()
(digi.xbee.devices.LPWANDevice method), 532
read_ip_data()
(digi.xbee.devices.NBIoTDevice method), 555
read_ip_data_from()
(digi.xbee.devices.CellularDevice method), 508
read_ip_data_from()
(digi.xbee.devices.DigiMeshDevice method), 388
read_ip_data_from()
(digi.xbee.devices.DigiPointDevice method), 414
read_ip_data_from()
(digi.xbee.devices.IPDevice method), 471
read_ip_data_from()
(digi.xbee.devices.LPWANDevice method), 533
read_ip_data_from()
(digi.xbee.devices.NBIoTDevice method), 556
read_ip_data_from()
(digi.xbee.devices.WiFiDevice method), 585
RemoteDigiPointDevice (class in digi.xbee.devices), 640
RemoteRaw802Device (class in digi.xbee.devices), 609
RemoteXBeedvice (class in digi.xbee.devices), 594
ReceivePacket (class in digi.xbee.models.options), 130
ReceiverPacket (class in digi.xbee.modelz.common), 171
ReceiveDevice (in module digi.xbee.recovery), 771
RecoveryException, 725
recv() (digi.xbee.xsocket.socket method), 775
recvfrom() (digi.xbee.xsocket.socket method), 776
register_joining_device() (digi.xbee.devices.ZigBeeDevice method), 441
register_joining_device() (digi.xbee.devices.ZigBeeDevice method), 441
register_device_status_packet (class in digi.xbee.packets.zigbee.RouteRecordIndicatorPacket), 317
RegisterDeviceStatusPacket (class in digi.xbee.packets.zigbee), 311
RegisterJoiningDevicePacket (class in digi.xbee.packets.zigbee), 308
RegisterKeyOptions (class in digi.xbee.models.options), 134
registrant_address (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket), 310
RelayDataReceived (class in digi.xbee.reader), 751
remote_address (digi.xbee.packets.socket.SocketNewIPv4ClientPacket), 289
remote_device (digi.xbee.models.message.ExplicitXBeeMessage method), 128
remote_device (digi.xbee.models.message.XBeeMessage attribute), 127
remote_port (digi.xbee.packets.socket.SocketNewIPv4ClientPacket), 289
RemoteATCmdOptions (class in digi.xbee.models.options), 132
RemoteATCommandPacket (class in digi.xbee.devices.common), 174
RemoteATCommandResponsePacket (class in digi.xbee.devices.common), 178
RemoteATCommandResponseWifiPacket (class in digi.xbee.packets.wifi), 305
RemoteATCommandWifiPacket (class in digi.xbee.packets.wifi), 302
RemoteDigiMeshDevice (class in digi.xbee.devices), 624
RemoteDigiPointDevice (class in digi.xbee.devices), 640
RemoteRaw802Device (class in digi.xbee.devices), 609
RemoteXBeedvice (class in digi.xbee.devices), 594
ReceivePacket (class in digi.xbee.models.options), 130
ReceiveDevice (in module digi.xbee.recovery), 771
RecoveryException, 725
recv() (digi.xbee.xsocket.socket method), 775
recvfrom() (digi.xbee.xsocket.socket method), 776
register_joining_device() (digi.xbee.devices.ZigBeeDevice method), 441
register_joining_device() (digi.xbee.devices.ZigBeeDevice method), 441
register_device_status_packet (class in digi.xbee.packets.zigbee.RouteRecordIndicatorPacket), 317
RegisterDeviceStatusPacket (class in digi.xbee.packets.zigbee), 311
RegisterJoiningDevicePacket (class in digi.xbee.packets.zigbee), 308
RegisterKeyOptions (class in digi.xbee.models.options), 134
registrant_address (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket), 310
RelayDataReceived (class in digi.xbee.reader), 751
remote_address (digi.xbee.packets.socket.SocketNewIPv4ClientPacket), 289
remote_device (digi.xbee.models.message.ExplicitXBeeMessage method), 128
remote_device (digi.xbee.models.message.XBeeMessage attribute), 127
remote_port (digi.xbee.packets.socket.SocketNewIPv4ClientPacket), 289
RemoteATCmdOptions (class in digi.xbee.models.options), 132
RemoteATCommandPacket (class in digi.xbee.devices.common), 174
RemoteATCommandResponsePacket (class in digi.xbee.devices.common), 178
RemoteATCommandResponseWifiPacket (class in digi.xbee.packets.wifi), 305
RemoteATCommandWifiPacket (class in digi.xbee.packets.wifi), 302
RemoteDigiMeshDevice (class in digi.xbee.devices), 624
RemoteDigiPointDevice (class in digi.xbee.devices), 640
RemoteRaw802Device (class in digi.xbee.devices), 609
RemoteXBeedvice (class in digi.xbee.devices), 594
ReceivePacket (class in digi.xbee.models.options), 130
ReceiveDevice (in module digi.xbee.recovery), 771
RecoveryException, 725
recv() (digi.xbee.xsocket.socket method), 775
recvfrom() (digi.xbee.xsocket.socket method), 776
register_joining_device() (digi.xbee.devices.ZigBeeDevice method), 441
register_joining_device() (digi.xbee.devices.ZigBeeDevice method), 441
register_device_status_packet (class in digi.xbee.packets.zigbee.RouteRecordIndicatorPacket), 317
RegisterDeviceStatusPacket (class in digi.xbee.packets.zigbee), 311
RegisterJoiningDevicePacket (class in digi.xbee.packets.zigbee), 308
RegisterKeyOptions (class in digi.xbee.models.options), 134
registrant_address (digi.xbee.packets.zigbee.RegisterJoiningDevicePacket), 310
RelayDataReceived (class in digi.xbee.reader), 751
remote_address (digi.xbeea.packets.socket.SocketNewIPv4ClientPacket), 289
remote_device (digi.xbee.models.message.ExplicitXBeeMessage method), 128
remote_device (digi.xbee.models.message.XBeeMessage attribute), 127
remote_port (digi.xbee.packets.socket.SocketNewIPv4ClientPacket), 289
RemoteATCmdOptions (class in digi.xbee.models.options), 132
RemoteATCommandPacket (class in digi.xbee.devices.common), 174
RemoteATCommandResponsePacket (class in digi.xbee.devices.common), 178
RemoteATCommandResponseWifiPacket (class in digi.xbee.packets.wifi), 305
RemoteATCommandWifiPacket (class in digi.xbee.packets.wifi), 302
RemoteDigiMeshDevice (class in digi.xbee.devices), 624
XBee Python Library Documentation, Release 1.3.0

method, 697
remove_device() (digixbee.devices.XBeeNetwork
method), 680
remove_device() (digixbee.devices.ZigBeeNetwork
method), 688
remove_element() (digixbee.filesystem.LocalXBeeFileManager
method), 727
REPEATER_MODE (digixbee.models.options.TransmitOptions
attribute), 132
request_data (digixbee.packets.devicecloud.DeviceRequestPacket
attribute), 202
request_data (digixbee.packets.devicecloud.DeviceResponsePacket
attribute), 205
request_id (digixbee.packets.devicecloud.DeviceRequestPacket
attribute), 202
reset() (digixbee.devices.AbstractXBeeDevice
method), 326
reset() (digixbee.devices.CellularDevice method), 508
reset() (digixbee.devices.DigiMeshDevice method), 403
reset() (digixbee.devices.DigiPointDevice method), 428
reset() (digixbee.devices.IPDevice method), 485
reset() (digixbee.devices.LPWANDevice method), 533
reset() (digixbee.devices.NBLoTDevice method), 556
reset() (digixbee.devices.Raw802Device method), 378
reset() (digixbee.devices.RemoteDigiMeshDevice
method), 634
reset() (digixbee.devices.RemoteDigiPointDevice
method), 649
reset() (digixbee.devices.RemoteRaw802Device
method), 618
reset() (digixbee.devices.RemoteXBeeDevice
method), 595
reset() (digixbee.devices.RemoteZigBeeDevice
method), 665
reset() (digixbee.devices.WiFiDevice method), 585
reset() (digixbee.devices.XBeeDevice method), 344
reset_settings (digixbee.profile.XBeeProfile
attribute), 459
reset_settings (digixbee.profile.XBeeProfile
attribute), 740
responder_addr (digixbee.packets.digimesh.RouteInformation
attribute), 221
response (digixbee.models.atcomm.ATCommandResponse
attribute), 118
reverse() (digixbee.reader.BluetoothDataReceived
method), 753
reverse() (digixbee.reader.DataReceived
method), 744
tribute), 244
rf_data (digi.xbee.packets.raw.RX64Packet attribute), 238
rf_data (digi.xbee.packets.raw.TX16Packet attribute), 222
rf_data (digi.xbee.packets.raw.TX64Packet attribute), 229
rf_data (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket attribute), 300
Role (class in digi.xbee.models.protocol), 136
RouteInformationPacket (class in digi.xbee.packets.digimesh), 217
RouteInformationReceived (class in digi.xbee.reader), 757
Run (class in digi.xbee.reader), 759
RX16IOPacket (class in digi.xbee.packets.raw), 245
RX16Packet (class in digi.xbee.packets.raw), 239
RX64IOPacket (class in digi.xbee.packets.raw), 242
RX64Packet (class in digi.xbee.packets.raw), 236
RXIPv4Packet (class in digi.xbee.packets.network), 222
RXSMSPacket (class in digi.xbee.packets.cellular), 157
S
scan_access_points ()
(digi.xbee.devices.WiFiDevice method), 566
scan_counter (digi.xbee.devices.AbstractXBeeDevice attribute), 338
scan_counter (digi.xbee.devices.CellularDevice attribute), 509
scan_counter (digi.xbee.devices.DigiMeshDevice attribute), 404
scan_counter (digi.xbee.devices.DigiMeshNetwork attribute), 707
scan_counter (digi.xbee.devices.DigiPointDevice attribute), 429
scan_counter (digi.xbee.devices.DigiPointNetwork attribute), 716
scan_counter (digi.xbee.devices.IPDevice attribute), 485
scan_counter (digi.xbee.devices.LPWANDevice attribute), 533
scan_counter (digi.xbee.devices.NBIoTDevice attribute), 556
scan_counter (digi.xbee.devices.Raw802Device attribute), 378
scan_counter (digi.xbee.devices.Raw802Network attribute), 697
scan_counter (digi.xbee.devices.RemoteDigiMeshDevice attribute), 634
scan_counter (digi.xbee.devices.RemoteDigiPointDevice attribute), 649
scan_counter (digi.xbee.devices.RemoteRaw802Device attribute), 619
scan_counter (digi.xbee.devices.RemoteXBeeDevice attribute), 604
scan_counter (digi.xbee.devices.RemoteZigBeeDevice attribute), 665
scan_counter (digi.xbee.devices.ZigBeeDevice attribute), 586
scan_counter (digi.xbee.devices.ZigBeeNetwork attribute), 356
scan_counter (digi.xbee.devices.ZigBeeNetwork attribute), 672
scan_counter_a2b (digi.xbee.devices.Connection attribute), 722
scan_counter_b2a (digi.xbee.devices.Connection attribute), 723
SCAN_TIL_CANCEL (digi.xbee.devices.XBeeNetwork attribute), 672
send ()
(digi.xbee.xsocket.socket method), 776
send_bluetooth_data ()
(digi.xbee.devices.CellularDevice method), 509
send_bluetooth_data ()
(digi.xbee.devices.DigiMeshDevice method), 404
send_bluetooth_data ()
(digi.xbee.devices.DigiPointDevice method), 429
send_bluetooth_data ()
(digi.xbee.devices.IPDevice method), 485
send_bluetooth_data ()
(digi.xbee.devices.LPWANDevice method), 533
send_bluetooth_data ()
(digi.xbee.devices.NBIoTDevice method), 556
send_bluetooth_data ()
(digi.xbee.devices.Raw802Device method),
send_data_broadcast()
(digi.xbee.devices.ZigBeeDevice method), 461
send_expl_data()
(digi.xbee.devices.DigiMeshDevice method), 388
send_expl_data()
(digi.xbee.devices.DigiPointDevice method), 414
send_expl_data()
(digi.xbee.devices.ZigBeeDevice method), 440
send_expl_data_async()
(digi.xbee.devices.DigiMeshDevice method), 389
send_expl_data_async()
(digi.xbee.devices.DigiPointDevice method), 414
send_expl_data_async()
(digi.xbee.devices.ZigBeeDevice method), 440
send_ip_data()
(digi.xbee.devices.CellularDevice method), 509
send_ip_data()
(digi.xbee.devices.IPDevice method), 470
send_ip_data()
(digi.xbee.devices.LPWANDevice method), 517
send_ip_data()
(digi.xbee.devices.NBIoTDevice method), 557
send_ip_data()
(digi.xbee.devices.WiFiDevice method), 586
send_ip_data_async()
(digi.xbee.devices.CellularDevice method), 510
send_ip_data_async()
(digi.xbee.devices.IPDevice method), 470
send_ip_data_async()
(digi.xbee.devices.LPWANDevice method), 517
send_ip_data_async()
(digi.xbee.devices.NBIoTDevice method), 557
send_ip_data_async()
(digi.xbee.devices.WiFiDevice method), 587
send_multicast_data()
(digi.xbee.devices.ZigBeeDevice method), 461
send_multicast_data_async()
(digi.xbee.devices.ZigBeeDevice method), 440
send_packet()
(digi.xbee.devices.CellularDevice method), 441
send_packet()
(digi.xbee.devices.DigiMeshDevice method), 511
send_packet()
(digi.xbee.devices.DigiPointDevice method), 406
send_packet()
(digi.xbee.devices.IPDevice method), 431
send_packet()
(digi.xbee.devices.LPWANDevice method), 534
send_packet()
(digi.xbee.devices.NBIoTDevice method), 557
send_packet()
(digi.xbee.devices.WiFiDevice method), 587
send_user_data_relay()
  (digi.xbee.devices.LPWANDevice method), 535
send_user_data_relay()
  (digi.xbee.devices.NBIoTDevice method), 559
send_user_data_relay()
  (digi.xbee.devices.Raw802Device method), 380
send_user_data_relay()
  (digi.xbee.devices.WiFiDevice method), 588
send_user_data_relay()
  (digi.xbee.devices.XBeeDevice method), 341
send_user_data_relay()
  (digi.xbee.devices.ZigBeeDevice method), 462
sendall () (digi.xbee.xsocket.socket method), 776
SendDataRequestOptions (class in digi.xbee.models.options), 133
SendDataRequestPacket (class in digi.xbee.packets.devicecloud), 212
SendDataResponsePacket (class in digi.xbee.packets.devicecloud), 215
sendto () (digi.xbee.xsocket.socket method), 777
SENT_FROM_END_DEVICE
  (digi.xbee.models.options.ReceiveOptions attribute), 130
serial_port (digi.xbee.devices.CellularDevice attribute), 512
serial_port (digi.xbee.devices.DigiMeshDevice attribute), 407
serial_port (digi.xbee.devices.DigiPointDevice attribute), 432
serial_port (digi.xbee.devices.IPDevice attribute), 486
serial_port (digi.xbee.devices.LPWANDevice attribute), 535
serial_port (digi.xbee.devices.NBIoTDevice attribute), 559
serial_port (digi.xbee.devices.Raw802Device attribute), 381
serial_port (digi.xbee.devices.WiFiDevice attribute), 588
serial_port (digi.xbee.devices.XBeeDevice attribute), 439
serial_port (digi.xbee.devices.ZigBeeDevice attribute), 462
set_16bit_addr () (digi.xbee.devices.AbstractXBeeDevice method), 327
set_16bit_addr () (digi.xbee.devices.CellularDevice method), 512
set_16bit_addr () (digi.xbee.devices.DigiMeshDevice method), 407
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Device Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.DigiPointDevice</code></td>
<td>432</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.IPDevice</code></td>
<td>487</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.LPWANDevice</code></td>
<td>535</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.NBLoTDevice</code></td>
<td>559</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.Raw802Device</code></td>
<td>381</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.RemoteDigiMeshDevice</code></td>
<td>634</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.RemoteDigiPointDevice</code></td>
<td>649</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.RemoteRaw802Device</code></td>
<td>619</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.RemoteXBeeDevice</code></td>
<td>604</td>
</tr>
<tr>
<td><code>set_16bit_addr()</code></td>
<td><code>digi.xbee.devices.RemoteZigBeeDevice</code></td>
<td>666</td>
</tr>
<tr>
<td><code>set_api_output_mode()</code></td>
<td><code>digi.xbee.devices.RemoteRaw802Device</code></td>
<td>619</td>
</tr>
<tr>
<td><code>set_api_output_mode()</code></td>
<td><code>digi.xbee.devices.WiFiDevice</code></td>
<td>589</td>
</tr>
<tr>
<td><code>set_api_output_mode()</code></td>
<td><code>digi.xbee.devices.XBeeDevice</code></td>
<td>357</td>
</tr>
<tr>
<td><code>set_api_output_mode()</code></td>
<td><code>digi.xbee.devices.ZigBeeDevice</code></td>
<td>462</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.AbstractXBeeDevice</code></td>
<td>336</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.CellularDevice</code></td>
<td>512</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.DigiMeshDevice</code></td>
<td>407</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.DigiPointDevice</code></td>
<td>432</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.IPDevice</code></td>
<td>487</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.LPWANDevice</code></td>
<td>536</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.NBLoTDevice</code></td>
<td>560</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.Raw802Device</code></td>
<td>382</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.RemoteDigiMeshDevice</code></td>
<td>635</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.RemoteDigiPointDevice</code></td>
<td>650</td>
</tr>
<tr>
<td><code>set_api_output_mode_value()</code></td>
<td><code>digi.xbee.devices.RemoteRaw802Device</code></td>
<td>619</td>
</tr>
</tbody>
</table>
set_api_output_mode_value()  
(digi.xbee.devices.RemoteXBeeDevice method), 605

set_api_output_mode_value()  
(digi.xbee.devices.RemoteZigBeeDevice method), 666

set_api_output_mode_value()  
(digi.xbee.devices.WiFiDevice method), 589

set_api_output_mode_value()  
(digi.xbee.devices.XBeeDevice method), 357

set_api_output_mode_value()  
(digi.xbee.devices.ZigBeeDevice method), 463

set_baudrate()  
(digi.xbee.serial.XBeeSerialPort method), 773

set_deep_discovery_options()  
(digi.xbee.devices.DigiMeshNetwork method), 707

set_deep_discovery_options()  
(digi.xbee.devices.DigiPointNetwork method), 716

set_deep_discovery_options()  
(digi.xbee.devices.Raw802Network method), 697

set_deep_discovery_options()  
(digi.xbee.devices.XBeeNetwork method), 677

set_deep_discovery_options()  
(digi.xbee.devices.ZigBeeNetwork method), 688

set_deep_discovery_timeouts()  
(digi.xbee.devices.DigiMeshNetwork method), 707

set_deep_discovery_timeouts()  
(digi.xbee.devices.DigiPointNetwork method), 717

set_deep_discovery_timeouts()  
(digi.xbee.devices.Raw802Network method), 698

set_deep_discovery_timeouts()  
(digi.xbee.devices.XBeeNetwork method), 679

set_deep_discovery_timeouts()  
(digi.xbee.devices.ZigBeeNetwork method), 688

set_dest_address()  
(digi.xbee.devices.AbstractXBeeDevice method), 329

set_dest_address()  
(digi.xbee.devices.CellularDevice method), 513

set_dest_address()  
(digi.xbee.devices.DigiPointDevice method), 433

set_dest_address()  
(digi.xbee.devices.IPDevice method), 472

set_dest_address()  
(digi.xbee.devices.LPWANDevice method), 536

set_dest_address()  
(digi.xbee.devices.NBLoTDevice method), 560

set_dest_address()  
(digi.xbee.devices.RemoteDigiMeshDevice method), 635

set_dest_address()  
(digi.xbee.devices.RemoteDigiPointDevice method), 650

set_dest_address()  
(digi.xbee.devices.RemoteRaw802Device method), 620

set_dest_address()  
(digi.xbee.devices.RemoteXBeeDevice method), 605

set_dest_address()  
(digi.xbee.devices.RemoteZigBeeDevice method), 667

set_dest_address()  
(digi.xbee.devices.WiFiDevice method), 590

set_dest_address()  
(digi.xbee.devices.XBeeDevice method), 358

set_dio_change_detection()  
(digi.xbee.devices.AbstractXBeeDevice method), 334

set_dest_ip_addr()  
(digi.xbee.devices.CellularDevice method), 513

set_dest_ip_addr()  
(digi.xbee.devices.IPDevice method), 469

set_dest_ip_addr()  
(digi.xbee.devices.LPWANDevice method), 537

set_dest_ip_addr()  
(digi.xbee.devices.NBLoTDevice method), 560

set_dest_ip_addr()  
(digi.xbee.devices.WiFiDevice method), 590

set_dest_ip_addr()  
(digi.xbee.devices.AbstractXBeeDevice method), 334
set_dio_change_detection()
(digi.xbee.devices.CellularDevice method), 495

set_dio_change_detection()
(digi.xbee.devices.DigiMeshDevice method), 408

set_dio_change_detection()
(digi.xbee.devices.DigiPointDevice method), 433

set_dio_change_detection()
(digi.xbee.devices.IPDevice method), 488

set_dio_change_detection()
(digi.xbee.devices.LPWANDevice method), 537

set_dio_change_detection()
(digi.xbee.devices.NBIoTDevice method), 561

set_dio_change_detection()
(digi.xbee.devices.Raw802Device method), 383

set_dio_change_detection()
(digi.xbee.devices.RemoteDigiMeshDevice method), 636

set_dio_change_detection()
(digi.xbee.devices.RemoteDigiPointDevice method), 651

set_dio_change_detection()
(digi.xbee.devices.RemoteRaw802Device method), 621

set_dio_change_detection()
(digi.xbee.devices.RemoteXBeeDevice method), 606

set_dio_change_detection()
(digi.xbee.devices.RemoteZigBeeDevice method), 667

set_dio_change_detection()
(digi.xbee.devices.XBeeDevice method), 358

set_dio_change_detection()
(digi.xbee.devices.ZigBeeDevice method), 464

set_discovery_options()
(digi.xbee.devices.DigiMeshNetwork method), 708

set_discovery_options()
(digi.xbee.devices.DigiPointNetwork method), 717

set_discovery_options()
(digi.xbee.devices.Raw802Network method), 699

set_discovery_options()
(digi.xbee.devices.XBeeNetwork method), 678

set_discovery_options()
(digi.xbee.devices.ZigBeeNetwork method), 689

set_discovery_timeout()
(digi.xbee.devices.DigiMeshNetwork method), 708

set_discovery_timeout()
(digi.xbee.devices.DigiPointNetwork method), 718

set_discovery_timeout()
(digi.xbee.devices.Raw802Network method), 699

set_discovery_timeout()
(digi.xbee.devices.XBeeNetwork method), 678

set_discovery_timeout()
(digi.xbee.devices.ZigBeeNetwork method), 689

set_dns_address()
(digi.xbee.devices.XBeeDevice method), 333

set_gateway_address()
(digi.xbee.devices.AbstractXBeeDevice method), 358

set_gateway_address()
(digi.xbee.devices.CellularDevice method), 513

set_gateway_address()
(digi.xbee.devices.DigiMeshDevice method), 409

set_gateway_address()
(digi.xbee.devices.DigiPointDevice method), 434

set_gateway_address()
(digi.xbee.devices.IPDevice method), 488

set_gateway_address()
(digi.xbee.devices.LPWANDevice method), 459

set_gateway_address()
(digi.xbee.devices.NBIoTDevice method), 537

set_gateway_address()
(digi.xbee.devices.Raw802Device method), 561

set_gateway_address()
(digi.xbee.devices.RemoteDigiMeshDevice method), 636

set_gateway_address()
(digi.xbee.devices.RemoteDigiPointDevice method), 651

set_gateway_address()
(digi.xbee.devices.RemoteRaw802Device method), 621

set_gateway_address()
(digi.xbee.devices.RemoteXBeeDevice method), 606

set_gateway_address()
(digi.xbee.devices.RemoteZigBeeDevice method), 667

set_gateway_address()
(digi.xbee.devices.XBeeDevice method), 358

set_gateway_address()
(digi.xbee.devices.ZigBeeDevice method), 464
set_io_configuration()
(digi.xbee.devices.AbstractXBeeDevice method), 330

set_io_configuration()
(digi.xbee.devices.CellularDevice method), 514

set_io_configuration()
(digi.xbee.devices.DigiMeshDevice method), 409

set_io_configuration()
(digi.xbee.devices.DigiPointDevice method), 434

set_io_configuration()
(digi.xbee.devices.IPDevice method), 489

set_io_configuration()
(digi.xbee.devices.LPWANDevice method), 538

set_io_configuration()
(digi.xbee.devices.NBIoTDevice method), 562

set_io_configuration()
(digi.xbee.devices.Raw802Device method), 383

set_io_configuration()
(digi.xbee.devices.RemoteDigiMeshDevice method), 636

set_io_configuration()
(digi.xbee.devices.RemoteDigiPointDevice method), 651

set_io_configuration()
(digi.xbee.devices.RemoteRaw802Device method), 621

set_io_configuration()
(digi.xbee.devices.RemoteXBeeDevice method), 607

set_io_configuration()
(digi.xbee.devices.RemoteZigBeeDevice method), 668

set_ip_address()
(digi.xbee.devices.WiFiDevice method), 569

set_ip_addressing_mode()
(digi.xbee.devices.WiFiDevice method), 569

set_local_xbee_device()
(digi.xbee.devices.RemoteDigiMeshDevice method), 637

set_local_xbee_device()
(digi.xbee.devices.RemoteDigiPointDevice method), 652

set_local_xbee_device()
(digi.xbee.devices.RemoteRaw802Device method), 622

set_local_xbee_device()
(digi.xbee.devices.RemoteXBeeDevice method), 606

set_sampling_rate()
(digi.xbee.devices.AbstractXBeeDevice method), 331

set_sampling_rate()
(digi.xbee.devices.CellularDevice method), 495

set_sampling_rate()
(digi.xbee.devices.DigiMeshDevice method), 409

set_sampling_rate()
(digi.xbee.devices.DigiPointDevice method), 434

set_sampling_rate()
(digi.xbee.devices.IPDevice method), 489

set_sampling_rate()
(digi.xbee.devices.LPWANDevice method), 538

set_sampling_rate()
(digi.xbee.devices.NBIoTDevice method), 562

set_sampling_rate()
(digi.xbee.devices.Raw802Device method), 384

set_sampling_rate()
(digi.xbee.devices.RemoteDigiMeshDevice method), 637

set_sampling_rate()
(digi.xbee.devices.RemoteDigiPointDevice method), 651

set_sampling_rate()
(digi.xbee.devices.RemoteRaw802Device method), 621

set_sampling_rate()
(digi.xbee.devices.RemoteXBeeDevice method), 607

set_sampling_rate()
(digi.xbee.devices.RemoteZigBeeDevice method), 668

set_sampling_rate()
(digi.xbee.devices.WiFiDevice method), 591

set_sampling_rate()
(digi.xbee.devices.XBeeDevice method), 359

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 465

set_sampling_rate()
(digi.xbee/devices.WiFiDevice method), 569
set_local_xbee_device() (digi.xbee.devices.RemoteZigBeeDevice method), 668
set_many_to_one_broadcasting_time() (digi.xbee.devices.ZigBeeDevice method), 439
set_mask_address() (digi.xbee.devices.WiFiDevice method), 570
set_node_id() (digi.xbee.devices.AbstractXBeeDevice method), 326
set_node_id() (digi.xbee.devices.CellularDevice method), 496
set_node_id() (digi.xbee.devices.DigiMeshDevice method), 410
set_node_id() (digi.xbee.devices.DigiPointDevice method), 435
set_node_id() (digi.xbee.devices.DigiMeshDevice method), 489
set_node_id() (digi.xbee.devices.IPDevice method), 472
set_node_id() (digi.xbee.devices.LPWANDevice method), 538
set_node_id() (digi.xbee.devices.NBIoTDevice method), 562
set_node_id() (digi.xbee.devices.Raw802Device method), 584
set_node_id() (digi.xbee.devices.RemoteDigiMeshDevice method), 637
set_node_id() (digi.xbee.devices.RemoteDigiPointDevice method), 652
set_node_id() (digi.xbee.devices.RemoteRaw802Device method), 622
set_node_id() (digi.xbee.devices.RemoteXBeeDevice method), 607
set_node_id() (digi.xbee.devices.RemoteZigBeeDevice method), 669
set_pan_id() (digi.xbee.devices.WeFiDevice method), 669
set_pan_id() (digi.xbee.devices.RemoteDigiMeshDevice method), 637
set_pan_id() (digi.xbee.devices.RemoteDigiPointDevice method), 652
set_pan_id() (digi.xbee.devices.RemoteRaw802Device method), 622
set_pan_id() (digi.xbee.devices.RemoteXBeeDevice method), 607
set_pan_id() (digi.xbee.devices.RemoteZigBeeDevice method), 669
set_parameter() (digi.xbee.devices.AbstractXBeeDevice method), 324
set_parameter() (digi.xbee.devices.CellularDevice method), 514
set_parameter() (digi.xbee.devices.DigiMeshDevice method), 410
set_parameter() (digi.xbee.devices.DigiPointDevice method), 435
set_parameter() (digi.xbee.devices.IPDevice method), 490
set_parameter() (digi.xbee.devices.DigiMeshDevice method), 490
set_parameter() (digi.xbee.devices.LPWANDevice method), 538
set_parameter() (digi.xbee.devices.NBIoTDevice method), 562
set_parameter() (digi.xbee.devices.RemoteDigiMeshDevice method), 637
set_parameter() (digi.xbee.devices.RemoteDigiPointDevice method), 652
set_parameter() (digi.xbee.devices.RemoteRaw802Device method), 622
set_power_level() (digi.xbee.devices.AbstractXBeeDevice method), 330
set_power_level() (digi.xbee.devices.Raw802Device method), 384
set_power_level() (digi.xbee.devices.RemoteDigiMeshDevice method), 637
set_power_level() (digi.xbee.devices.RemoteDigiPointDevice method), 652
set_power_level() (digi.xbee.devices.RemoteRaw802Device method), 622
set_power_level() (digi.xbee.devices.RemoteXBeeDevice method), 595
set_power_level() (digi.xbee.devices.RemoteZigBeeDevice method), 669
set_power_level() (digi.xbee.devices.DigiMeshDevice method), 410
set_power_level() (digi.xbee.devices.DigiPointDevice method), 435
set_power_level() (digi.xbee.devices.IPDevice method), 472
set_power_level() (digi.xbee.devices.LPWANDevice method), 538
set_power_level() (digi.xbee.devices.NBIoTDevice method), 562
set_power_level()
(digi.xbee.devices.CellularDevice method), 496

set_power_level()
(digi.xbee.devices.DigiMeshDevice method), 410

set_power_level()
(digi.xbee.devices.DigiPointDevice method), 435

set_power_level()
(digi.xbee.devices.IPDevice method), 490

set_power_level()
(digi.xbee.devices.LPWANDevice method), 538

set_power_level()
(digi.xbee.devices.NBIoTDevice method), 562

set_power_level()
(digi.xbee.devices.Raw802Device method), 384

set_power_level()
(digi.xbee.devices.RemoteDigiMeshDevice method), 638

set_power_level()
(digi.xbee.devices.RemoteDigiPointDevice method), 653

set_power_level()
(digi.xbee.devices.RemoteRaw802Device method), 622

set_power_level()
(digi.xbee.devices.RemoteXBeeDevice method), 607

set_power_level()
(digi.xbee.devices.RemoteZigBeeDevice method), 669

set_read_timeout()
(digi.xbee.serial.XBeeSerialPort method), 773

set_sync_ops_timeout()
(digi.xbee.devices.AbstractXBeeDevice method), 328

set_sync_ops_timeout()
(digi.xbee.devices.CellularDevice method), 515

set_sync_ops_timeout()
(digi.xbee.devices.DigiMeshDevice method), 411

set_sync_ops_timeout()
(digi.xbee.devices.DigiPointDevice method), 436

set_sync_ops_timeout()
(digi.xbee.devices.IPDevice method), 490

set_sync_ops_timeout()
(digi.xbee.devices.LPWANDevice method), 539

set_sync_ops_timeout()
set_sync_ops_timeout()
(digi.xbee.devices.NBIoTDevice method), 563
socket_id (digi.xbee.packets.socket.SocketNewIPv4ClientPacket attribute), 289
set_sync_ops_timeout()
(digi.xbee.devices.Raw802Device method), 385
socket_id (digi.xbee.packets.socket.SocketOptionRequestPacket attribute), 260
set_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiMeshDevice method), 638
socket_id (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
set_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiPointDevice method), 653
socket_id (digi.xbee.packets.socket.SocketReceivePacket attribute), 292
set_sync_ops_timeout()
(digi.xbee.devices.RemoteRaw802Device method), 623
socket_id (digi.xbee.packets.socket.SocketSendPacket attribute), 278
set_sync_ops_timeout()
(digi.xbee.devices.RemoteXBeeDevice method), 608
socket_id (digi.xbee.packets.socket.SocketSendToPacket attribute), 281
set_sync_ops_timeout()
(digi.xbee.devices.RemoteZigBeeDevice method), 670
SocketBindListenPacket (class in digi.xbee.packets.socket), 282
setblocking() (digi.xbee.xsocket.socket method), 775
SocketClosePacket (class in digi.xbee.packets.socket), 271
setsockopt() (digi.xbee.xsocket.socket method), 777
SocketCloseResponsePacket (class in digi.xbee.packets.socket), 274
settimeout() (digi.xbee.xsocket.socket method), 777
SocketConnectPacket (class in digi.xbee.packets.socket), 265
signal_quality (digi.xbee.models.accesspoint.AccessPoint attribute), 115
SocketConnectResponsePacket (class in digi.xbee.packets.socket), 268
size (digi.xbee.filesystem.FileSystemElement attribute), 726
SocketCreatePacket (class in digi.xbee.packets.socket), 254
SMSMessage (class in digi.xbee.models.message), 129
SocketCreateResponsePacket (class in digi.xbee.packets.socket), 256
SMSSaved (class in digi.xbee.models.message), 129
SocketDataReceived (class in digi.xbee.reader), 755
SMSReceived (class in digi.xbee.models.message), 129
SocketDataReceivedFrom (class in digi.xbee.reader), 755
socket (class in digi.xbee.xsocket), 774
SocketInfoState (class in digi.xbee.models.status), 146
socket_id (digi.xbee.packets.socket.SocketBindListenPacket attribute), 283
SocketListenResponsePacket (class in digi.xbee.packets.socket), 285
socket_id (digi.xbee.packets.socket.SocketClosePacket attribute), 272
SocketNewIPv4ClientPacket (class in digi.xbee.packets.socket), 288
socket_id (digi.xbee.packets.socket.SocketCloseResponsePacket attribute), 275
socket (class in digi.xbee.xsocket), 774
socket_id (digi.xbee.packets.socket.SocketConnectPacket attribute), 267
socket_id (digi.xbee.packets.socket.SocketCreatePacket attribute), 258
socket_id (digi.xbee.packets.socket.SocketDataReceivedPacket attribute), 286
socket_id (digi.xbee.packets.socket.SocketDataReceivedFromPacket attribute), 293
socket_id (digi.xbee.packets.socket.SocketDataSentPacket attribute), 269
socket_id (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 281
socket_id (digi.xbee.packets.socket.SocketMessagePacket attribute), 259
socket_id (digi.xbee.packets.socket.SocketNewIPv4ClientPacket attribute), 289
socket_id (digi.xbee.packets.socket.SocketOptionPacket attribute), 262
socket_id (digi.xbee.packets.socket.SocketOptionRequestPacket attribute), 260
socket_id (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
signal_quality (digi.xbee.models.accesspoint.AccessPoint attribute), 115
size (digi.xbee.filesystem.FileSystemElement attribute), 726
SMSMessage (class in digi.xbee.models.message), 129
SMSSaved (class in digi.xbee.models.message), 129
socket (class in digi.xbee.xsocket), 774
socket_id (digi.xbee.packets.socket.SocketBindListenPacket attribute), 283
socket_id (digi.xbee.packets.socket.SocketClosePacket attribute), 272
socket_id (digi.xbee.packets.socket.SocketCloseResponsePacket attribute), 275
socket_id (digi.xbee.packets.socket.SocketConnectPacket attribute), 267
socket_id (digi.xbee.packets.socket.SocketCreatePacket attribute), 258
socket_id (digi.xbee.packets.socket.SocketDataReceivedPacket attribute), 286
socket_id (digi.xbee.packets.socket.SocketDataReceivedFromPacket attribute), 293
socket_id (digi.xbee.packets.socket.SocketDataSentPacket attribute), 269
socket_id (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 281
socket_id (digi.xbee.packets.socket.SocketMessagePacket attribute), 259
socket_id (digi.xbee.packets.socket.SocketNewIPv4ClientPacket attribute), 289
socket_id (digi.xbee.packets.socket.SocketOptionPacket attribute), 262
socket_id (digi.xbee.packets.socket.SocketOptionRequestPacket attribute), 260
socket_id (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
signal_quality (digi.xbee.models.accesspoint.AccessPoint attribute), 115
size (digi.xbee.filesystem.FileSystemElement attribute), 726
SMSMessage (class in digi.xbee.models.message), 129
SMSSaved (class in digi.xbee.models.message), 129
socket (class in digi.xbee.xsocket), 774
socket_id (digi.xbee.packets.socket.SocketBindListenPacket attribute), 283
socket_id (digi.xbee.packets.socket.SocketClosePacket attribute), 272
socket_id (digi.xbee.packets.socket.SocketCloseResponsePacket attribute), 275
socket_id (digi.xbee.packets.socket.SocketConnectPacket attribute), 267
socket_id (digi.xbee.packets.socket.SocketCreatePacket attribute), 258
socket_id (digi.xbee.packets.socket.SocketDataReceivedPacket attribute), 286
socket_id (digi.xbee.packets.socket.SocketDataReceivedFromPacket attribute), 293
socket_id (digi.xbee.packets.socket.SocketDataSentPacket attribute), 269
socket_id (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 281
socket_id (digi.xbee.packets.socket.SocketMessagePacket attribute), 259
socket_id (digi.xbee.packets.socket.SocketNewIPv4ClientPacket attribute), 289
socket_id (digi.xbee.packets.socket.SocketOptionPacket attribute), 262
socket_id (digi.xbee.packets.socket.SocketOptionRequestPacket attribute), 260
socket_id (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
signal_quality (digi.xbee.models.accesspoint.AccessPoint attribute), 115
size (digi.xbee.filesystem.FileSystemElement attribute), 726
SMSMessage (class in digi.xbee.models.message), 129
SMSSaved (class in digi.xbee.models.message), 129
socket (class in digi.xbee.xsocket), 774
SocketState (class in digi.xbee.models.status), 146
SocketStatePacket (class in digi.xbee.packets.socket), 296
SocketStateReceived (class in digi.xbee.reader), 754
SocketStatus (class in digi.xbee.models.status), 145

SpecialByte (class in digi.xbee.models.atcomm), 117
src_addr (digi.xbee.packets.digimesh.RouteInformationPacket attribute), 221
src_event (digi.xbee.packets.digimesh.RouteInformationPacket attribute), 219
src_interface (digi.xbee.packets.relays.UserDataRelayOutputPacket attribute), 253
ssid (digi.xbee.models.accesspoint.AccessPoint attribute), 115
start() (digi.xbee.reader.PacketListener method), 769
start_discovery_process() (digi.xbee.devices.DigiMeshNetwork method), 709
start_discovery_process() (digi.xbee.devices.DigiPointNetwork method), 718
start_discovery_process() (digi.xbee.devices.Raw802Network method), 699
start_discovery_process() (digi.xbee.devices.XBeeNetwork method), 672
start_discovery_process() (digi.xbee.devices.ZigBeeNetwork method), 690
start_listening() (digi.xbee.devices.CellularDevice method), 515
start_listening() (digi.xbee.devices.IPDevice method), 469
start_listening() (digi.xbee.devices.LPWANDevice method), 563
start_listening() (digi.xbee.devices.WiFiDevice method), 593
state (digi.xbee.packets.socket.SocketStatePacket attribute), 299
status (digi.xbee.models.atcomm.ATCommandResponse attribute), 118
status (digi.xbee.packets.common.ATCommResponsePacket attribute), 170
status (digi.xbee.packets.common.RemoteATCommandResponsePacket attribute), 179
status (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket attribute), 208
status (digi.xbee.packets.devicecloud.SendDataResponsePacket attribute), 217
status (digi.xbee.packets.socket.SocketCloseResponsePacket attribute), 275
status (digi.xbee.packets.socket.SocketConnectResponsePacket attribute), 269
status (digi.xbee.packets.socket.SocketCreateResponsePacket attribute), 258
status (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 286
status (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
status (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket attribute), 308
status_a2b (digi.xbee.devices.Connection attribute), 722
status_b2a (digi.xbee.devices.Connection attribute), 722
stop() (digi.xbee.reader.PacketListener method), 762
stop_bits (digi.xbee.profile.FirmwareStopbits attribute), 737
stop_discovery_process() (digi.xbee.devices.DigiMeshNetwork method), 709
stop_discovery_process() (digi.xbee.devices.DigiPointNetwork method), 719
stop_discovery_process() (digi.xbee.devices.Raw802Network method), 700
stop_discovery_process() (digi.xbee.devices.XBeeNetwork method), 673
stop_discovery_process() (digi.xbee.devices.ZigBeeNetwork method), 690
stop_listening() (digi.xbee.devices.CellularDevice method), 515
stop_listening() (digi.xbee.devices.IPDevice method), 470
stop_listening() (digi.xbee.devices.LPWANDevice method), 539
stop_listening() (digi.xbee.devices.NBIoTDevice method), 563
stop_listening() (digi.xbee.devices.WiFiDevice method), 593
successor_addr (digi.xbee.packets.digimesh.RouteInformationPacket attribute), 222
tag (digi.xbee.profile.XBeeSettingFormat attribute), 738
tag (digi.xbee.profile.XBeeSettingType attribute), 738
to_dict() (digi.xbee.models.message.ExplicitXBeeMessage method), 128
to_dict() (digi.xbee.models.message.IPMessage method), 128
to_dict() (digi.xbee.models.message.SMSMessage method), 129
to_dict() (digi.xbee.models.message.UserDataRelayMessage method), 130
to_dict() (digi.xbee.models.message.XBeeMessage method), 127
to_dict() (digi.xbee.packets.base.GenericXBeePacket method), 154
to_dict() (digi.xbee.packets.base.UnknownXBeePacket method), 157
to_dict() (digi.xbee.packets.base.XBeeAPIPacket method), 152
to_dict() (digi.xbee.packets.cellular.RXSMSPacket method), 154
to_dict() (digi.xbee.packets.cellular.TXSMSPacket method), 160
status (digi.xbee.packets.socket.SocketStatePacket attribute), 299
status (digi.xbee.models.atcomm.ATCommandResponse attribute), 118
status (digi.xbee.packets.common.ATCommResponsePacket attribute), 170
status (digi.xbee.packets.common.RemoteATCommandResponsePacket attribute), 179
status (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket attribute), 208
status (digi.xbee.packets.devicecloud.SendDataResponsePacket attribute), 217
status (digi.xbee.packets.socket.SocketCloseResponsePacket attribute), 275
status (digi.xbee.packets.socket.SocketConnectResponsePacket attribute), 269
status (digi.xbee.packets.socket.SocketCreateResponsePacket attribute), 258
status (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 286
status (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
status (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket attribute), 308
status_a2b (digi.xbee.devices.Connection attribute), 722
status_b2a (digi.xbee.devices.Connection attribute), 722
stop() (digi.xbee.reader.PacketListener method), 762
stop_bits (digi.xbee.profile.FirmwareStopbits attribute), 737
stop_discovery_process() (digi.xbee.devices.DigiMeshNetwork method), 709
stop_discovery_process() (digi.xbee.devices.DigiPointNetwork method), 719
stop_discovery_process() (digi.xbee.devices.Raw802Network method), 700
stop_discovery_process() (digi.xbee.devices.XBeeNetwork method), 673
stop_discovery_process() (digi.xbee.devices.ZigBeeNetwork method), 690
stop_listening() (digi.xbee.devices.CellularDevice method), 515
stop_listening() (digi.xbee.devices.IPDevice method), 470
stop_listening() (digi.xbee.devices.LPWANDevice method), 539
stop_listening() (digi.xbee.devices.NBIoTDevice method), 563
stop_listening() (digi.xbee.devices.WiFiDevice method), 593
successor_addr (digi.xbee.packets.digimesh.RouteInformationPacket attribute), 222
tag (digi.xbee.profile.XBeeSettingFormat attribute), 738
tag (digi.xbee.profile.XBeeSettingType attribute), 738
to_dict() (digi.xbee.models.message.ExplicitXBeeMessage method), 128
to_dict() (digi.xbee.models.message.IPMessage method), 128
to_dict() (digi.xbee.models.message.SMSMessage method), 129
to_dict() (digi.xbee.models.message.UserDataRelayMessage method), 130
to_dict() (digi.xbee.models.message.XBeeMessage method), 127
to_dict() (digi.xbee.packets.base.GenericXBeePacket method), 154
to_dict() (digi.xbee.packets.base.UnknownXBeePacket method), 157
to_dict() (digi.xbee.packets.base.XBeeAPIPacket method), 152
to_dict() (digi.xbee.packets.cellular.RXSMSPacket method), 154
to_dict() (digi.xbee.packets.cellular.TXSMSPacket method), 160
status (digi.xbee.packets.socket.SocketStatePacket attribute), 299
status (digi.xbee.models.atcomm.ATCommandResponse attribute), 118
status (digi.xbee.packets.common.ATCommResponsePacket attribute), 170
status (digi.xbee.packets.common.RemoteATCommandResponsePacket attribute), 179
status (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket attribute), 208
status (digi.xbee.packets.devicecloud.SendDataResponsePacket attribute), 217
status (digi.xbee.packets.socket.SocketCloseResponsePacket attribute), 275
status (digi.xbee.packets.socket.SocketConnectResponsePacket attribute), 269
status (digi.xbee.packets.socket.SocketCreateResponsePacket attribute), 258
status (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 286
status (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
status (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket attribute), 308
status_a2b (digi.xbee.devices.Connection attribute), 722
status_b2a (digi.xbee.devices.Connection attribute), 722
stop() (digi.xbee.reader.PacketListener method), 762
stop_bits (digi.xbee.profile.FirmwareStopbits attribute), 737
stop_discovery_process() (digi.xbee.devices.DigiMeshNetwork method), 709
stop_discovery_process() (digi.xbee.devices.DigiPointNetwork method), 719
stop_discovery_process() (digi.xbee.devices.Raw802Network method), 700
stop_discovery_process() (digi.xbee.devices.XBeeNetwork method), 673
stop_discovery_process() (digi.xbee.devices.ZigBeeNetwork method), 690
stop_listening() (digi.xbee.devices.CellularDevice method), 515
stop_listening() (digi.xbee.devices.IPDevice method), 470
stop_listening() (digi.xbee.devices.LPWANDevice method), 539
stop_listening() (digi.xbee.devices.NBIoTDevice method), 563
stop_listening() (digi.xbee.devices.WiFiDevice method), 593
successor_addr (digi.xbee.packets.digimesh.RouteInformationPacket attribute), 222
tag (digi.xbee.profile.XBeeSettingFormat attribute), 738
tag (digi.xbee.profile.XBeeSettingType attribute), 738
to_dict() (digi.xbee.models.message.ExplicitXBeeMessage method), 128
to_dict() (digi.xbee.models.message.IPMessage method), 128
to_dict() (digi.xbee.models.message.SMSMessage method), 129
to_dict() (digi.xbee.models.message.UserDataRelayMessage method), 130
to_dict() (digi.xbee.models.message.XBeeMessage method), 127
to_dict() (digi.xbee.packets.base.GenericXBeePacket method), 154
to_dict() (digi.xbee.packets.base.UnknownXBeePacket method), 157
to_dict() (digi.xbee.packets.base.XBeeAPIPacket method), 152
to_dict() (digi.xbee.packets.cellular.RXSMSPacket method), 154
to_dict() (digi.xbee.packets.cellular.TXSMSPacket method), 160
status (digi.xbee.packets.socket.SocketStatePacket attribute), 299
status (digi.xbee.models.atcomm.ATCommandResponse attribute), 118
status (digi.xbee.packets.common.ATCommResponsePacket attribute), 170
status (digi.xbee.packets.common.RemoteATCommandResponsePacket attribute), 179
status (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket attribute), 208
status (digi.xbee.packets.devicecloud.SendDataResponsePacket attribute), 217
status (digi.xbee.packets.socket.SocketCloseResponsePacket attribute), 275
status (digi.xbee.packets.socket.SocketConnectResponsePacket attribute), 269
status (digi.xbee.packets.socket.SocketCreateResponsePacket attribute), 258
status (digi.xbee.packets.socket.SocketListenResponsePacket attribute), 286
status (digi.xbee.packets.socket.SocketOptionResponsePacket attribute), 263
status (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket attribute), 308
status_a2b (digi.xbee.devices.Connection attribute), 722
status_b2a (digi.xbee.devices.Connection attribute), 722
stop() (digi.xbee.reader.PacketListener method), 762
stop_bits (digi.xbee.profile.FirmwareStopbits attribute), 737
stop_discovery_process() (digi.xbee.devices.DigiMeshNetwork method), 709
stop_discovery_process() (digi.xbee.devices.DigiPointNetwork method), 719
stop_discovery_process() (digi.xbee.devices.Raw802Network method), 700
stop_discovery_process() (digi.xbee.devices.XBeeNetwork method), 673
stop_discovery_process() (digi.xbee.devices.ZigBeeNetwork method), 690
stop_listening() (digi.xbee.devices.CellularDevice method), 515
stop_listening() (digi.xbee.devices.IPDevice method), 470

transmit_options (digixbee.packets.raw.TX16Packet unescape_data() (digixbee.packets.common.ATCommResponsePacket
static method), 232
transmit_options (digixbee.packets.raw.TX64Packet unescape_data() (digixbee.packets.common.ExplicitAddressingPacket
static method), 197
transmit_options (digixbee.packets.wifi.RemoteATCommandWifiPacket () (digixbee.packets.common.ExplicitRXIndicatorPacket
static method), 199
transmit_status (digixbee.packets.common.TransmitStatusPacket
attribute), 186
transmit_status (digixbee.packets.common.TransmitStatusPacket
attribute), 186
transmit_status (digixbee.packets.raw.TXStatusPacket
attribute), 235
TransmitException, 724
TransmitOptions (class in digixbee.models.options), 131
TransmitPacket (class in digixbee.packets.common), 181
TransmitStatus (class in digixbee.models.status), 138
TransmitStatusPacket (class in digixbee.packets.common), 184
transport (digixbee.packets.devicecloud.DeviceRequestPacket
attribute), 202
TX16Packet (class in digixbee.packets.raw), 231
TX64Packet (class in digixbee.packets.raw), 228
tx_block_count (digixbee.packets.digimesh.RouteInformationPacket
attribute), 219
TXIPv4Packet (class in digixbee.packets.network), 225
TXSMSPacket (class in digixbee.packets.cellular), 160
TXStatusPacket (class in digixbee.packets.raw), 234
type (digixbee.profile.XBeeProfileSetting attribute), 739

U
unescape_data() (digixbee.packets.base.GenericXBeePacket
static method), 155
unescape_data() (digixbee.packets.base.UnknownXBeePacket
static method), 157
unescape_data() (digixbee.packets.base.XBeeAPIPacket
static method), 152
unescape_data() (digixbee.packets.base.XBeePacket
static method), 150
unescape_data() (digixbee.packets.cellular.RXSMSPacket
static method), 160
unescape_data() (digixbee.packets.cellular.TXSMSPacket
static method), 162
unescape_data() (digixbee.packets.common.ATCommPacket
static method), 165
unescape_data() (digixbee.packets.common.ATCommQueuePacket
static method), 168

Index
851
unescape_data() (digi.xbee.packets.relay.UserDataRelayPacket digi.xbee.packets.base), 155
static method), 251
unregister_joining_device() (digi.xbee.devices.ZigBeeDevice method), 443
unregister_joining_device_async() (digi.xbee.devices.ZigBeeDevice method), 273
update_bluetooth_password() (digi.xbee.devices.AbstractXBeeDevice method), 337
update_bluetooth_password() (digi.xbee.devices.CellularDevice method), 515
update_bluetooth_password() (digi.xbee.devices.DigiMeshDevice method), 411
update_bluetooth_password() (digi.xbee.devices.DigiPointDevice method), 436
update_bluetooth_password() (digi.xbee.devices.IPDevice method), 490
update_bluetooth_password() (digi.xbee.devices.LPWANDevice method), 539
update_bluetooth_password() (digi.xbee.devices.NBiotDevice method), 563
update_bluetooth_password() (digi.xbee.devices.RemoteDigiMeshDevice method), 638
update_bluetooth_password() (digi.xbee.devices.RemoteDigiPointDevice method), 653
update_bluetooth_password() (digi.xbee.devices.RemoteRaw802Device method), 623
update_bluetooth_password() (digi.xbee.devices.RemoteXBeeDevice method), 608
update_bluetooth_password() (digi.xbee.devices.RemoteZigBeeDevice method), 670
update_bluetooth_password() (digi.xbee.devices.WiFiDevice method), 593
UnknownXBeePacket (class in update_device_data_from())
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.AbstractXBeeDevice method)</td>
<td>324</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.CellularDevice method)</td>
<td>515</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.DigiMeshDevice method)</td>
<td>411</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.DigiPointDevice method)</td>
<td>436</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.IPDevice method)</td>
<td>491</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.LPWANDevice method)</td>
<td>539</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.NBLoTDevice method)</td>
<td>563</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.Raw802Device method)</td>
<td>385</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.RemoteDigiMeshDevice method)</td>
<td>639</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.RemoteDigiPointDevice method)</td>
<td>654</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.RemoteRaw802Device method)</td>
<td>623</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.RemoteXBeeDevice method)</td>
<td>623</td>
</tr>
<tr>
<td>update_device_data_from()</td>
<td>(digi.xbee.devices.ZigBeeDevice method)</td>
<td>467</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.DigiPointDevice method)</td>
<td>436</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.IPDevice method)</td>
<td>491</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.LPWANDevice method)</td>
<td>539</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.NBLoTDevice method)</td>
<td>563</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.Raw802Device method)</td>
<td>385</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.RemoteDigiMeshDevice method)</td>
<td>639</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.RemoteDigiPointDevice method)</td>
<td>654</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.RemoteRaw802Device method)</td>
<td>623</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.RemoteXBeeDevice method)</td>
<td>608</td>
</tr>
<tr>
<td>update_firmware()</td>
<td>(digi.xbee.devices.ZoomBeeDevice method)</td>
<td>670</td>
</tr>
<tr>
<td>update_local_firmware()</td>
<td>(module)</td>
<td>729</td>
</tr>
<tr>
<td>update_remote_firmware()</td>
<td>(module)</td>
<td>730</td>
</tr>
<tr>
<td>UpdateProfileException</td>
<td></td>
<td>739</td>
</tr>
<tr>
<td>USE_BROADCAST_PAN_ID</td>
<td>(digi.xbee.models.options.TransmitOptions attribute)</td>
<td>131</td>
</tr>
<tr>
<td>USE_EXTENDED_TIMEOUT</td>
<td>(digi.xbee.models.options.TransmitOptions attribute)</td>
<td>132</td>
</tr>
<tr>
<td>UserDataRelayMessage</td>
<td>(class)</td>
<td>129</td>
</tr>
<tr>
<td>UserDataRelayOutputPacket</td>
<td>(class)</td>
<td>251</td>
</tr>
<tr>
<td>UserDataRelayPacket</td>
<td>(class)</td>
<td>248</td>
</tr>
</tbody>
</table>
V
value (digi.xbee.profile.XBeeProfileSetting attribute), 739
version (digi.xbee.profile.XBeeProfile attribute), 740

W
wait_for_frame() (digi.xbee.comm_interface.XBeeCommunicationInterface
method), 323
wait_for_frame() (digi.xbee.serial.XBeeSerialPort
method), 773
wait_until_started()
(digi.xbee.reader.PacketListener method), 761
WiFiAssociationIndicationStatus (class in
digi.xbee.models.status), 144
WiFiDevice (class in digi.xbee.devices), 564
WiFiEncryptionType (class in
digi.xbee.models.accesspoint), 115
with_traceback()
(digi.xbee.exception.ATCommandException
method), 723
with_traceback()
(digi.xbee.exception.CommunicationException
method), 723
with_traceback()
(digi.xbee.exception.ConnectionException
method), 723
with_traceback()
(digi.xbee.exception.FirmwareUpdateException
method), 724
with_traceback()
(digi.xbee.exception.InvalidConfigurationException
method), 724
with_traceback()
(digi.xbee.exception.InvalidPacketException
method), 724
with_traceback()
(digi.xbee.exception.OperationNotSupportedException
method), 724
wait_until_started() (digi.xbee.reader.PacketListener
method), 761
x16bit_dest_addr (digi.xbee.packets.common.ExplicitAddressingPacket
attribute), 195
x16bit_dest_addr (digi.xbee.packets.common.RemoteATCommandPacket
attribute), 176
x16bit_dest_addr (digi.xbee.packets.common.TransmitPacket
attribute), 183
x16bit_dest_addr (digi.xbee.packets.common.TransmitStatusPacket
attribute), 186
x16bit_dest_addr (digi.xbee.packets.raw.TX16Packet
attribute), 232
x16bit_dest_addr (digi.xbee.packets.common.ExplicitRXIndicatorPacket
attribute), 200
x16bit_dest_addr (digi.xbee.packets.common.IODataSampleRxIndicatorPacket
attribute), 192