1 Requirements

2 Contents

2.1 Getting Started ........................................... 5
2.2 User Documentation ....................................... 5
2.3 Examples ..................................................... 6
2.4 FAQ ........................................................ 6
2.5 API reference ............................................... 6
  2.5.1 Get started with XBee Python library .................. 6
    2.5.1.1 Install your software ............................... 6
    2.5.1.2 Configure your XBee modules ...................... 8
    2.5.1.3 Run your first XBee Python application .......... 11
  2.5.2 XBee terminology ...................................... 14
    2.5.2.1 RF modules ........................................ 14
    2.5.2.2 XBee RF modules .................................. 14
    2.5.2.3 Radio firmware ..................................... 14
    2.5.2.4 Radio communication protocols .................... 14
    2.5.2.5 Radio module operating modes .................... 15
    2.5.2.6 API frames .......................................... 17
    2.5.2.7 AT settings or commands ........................... 18
  2.5.3 Work with XBee classes ................................. 18
    2.5.3.1 Instantiate an XBee device ....................... 19
    2.5.3.2 Open the XBee device connection .................. 20
    2.5.3.3 Close the XBee device connection ................. 23
  2.5.4 Configure the XBee device .............................. 24
    2.5.4.1 Read and set common parameters ................... 24
    2.5.4.2 Read, set and execute other parameters .......... 26
    2.5.4.3 Apply configuration changes ....................... 28
    2.5.4.4 Write configuration changes ....................... 29
    2.5.4.5 Reset the device .................................. 30
    2.5.4.6 Configure Wi-Fi settings .......................... 31
    2.5.4.7 Configure Bluetooth settings ..................... 33
  2.5.5 Discover the XBee network ............................. 35
    2.5.5.1 Configure the discovery process ................... 36
    2.5.5.2 Discover the network ............................... 37
    2.5.5.3 Access the discovered devices ..................... 39
2.5.4 Add and remove devices manually ........................................ 40
2.5.6 Communicate with XBee devices ........................................ 42
  2.5.6.1 Send and receive data ........................................... 43
  2.5.6.2 Send and receive explicit data ................................... 50
  2.5.6.3 Send and receive IP data ....................................... 58
  2.5.6.4 Send and receive SMS messages .................................. 64
  2.5.6.5 Send and receive Bluetooth data ................................ 67
  2.5.6.6 Send and receive MicroPython data ............................... 69
  2.5.6.7 Receive modem status events .................................... 71
2.5.7 Handle analog and digital IO lines .................................... 72
  2.5.7.1 Configure the IO lines .......................................... 72
  2.5.7.2 Read IO samples .............................................. 77
  2.5.7.3 Change detection sampling ..................................... 79
2.5.8 Log events ............................................................ 81
  2.5.8.1 Logging level ................................................... 83
2.5.9 XBee Python samples .................................................. 84
  2.5.9.1 Configuration samples .......................................... 84
  2.5.9.2 Network samples .............................................. 85
  2.5.9.3 Communication samples ....................................... 86
  2.5.9.4 IO samples .................................................... 91
2.5.10 Frequently Asked Questions (FAQs) ................................ 92
  2.5.10.1 What is XCTU and how do I download it? ....................... 92
  2.5.10.2 How do I find the serial port and baud rate of my module? 93
  2.5.10.3 Can I use the XBee Python Library with modules in AT operating mode? 94
  2.5.10.4 I get the Python error ImportError: No module named 'serial' 94
  2.5.10.5 I get the Python error ImportError: No module named 'arp' 94
2.5.11 API reference ........................................................ 94
  2.5.11.1 digi package .................................................. 94

3  Indices and tables ......................................................... 553
4  License ............................................................................ 555

Python Module Index .......................................................... 557
Index .................................................................................. 559
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*
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](https://pypi.org/project/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](https://pypi.org/project/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](https://pypi.org/project/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** in 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 6E 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>XBe...</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
- Cellular NB-IoT
- 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 LSB</td>
<td>API-specific Structure</td>
<td>1 Byte</td>
</tr>
</tbody>
</table>

**Start delimiter**
This field is always 0x7E.

**Length**
The length field has a two-byte value that specifies the number of bytes that are contained in the frame data field. It does not include the checksum field.

**Frame Data**
The content of this field is composed by the API identifier and the API identifier specific data. Depending on the API identifier (also called API frame type), the content of the specific data changes.

**Checksum**
Byte containing the hash sum of the API frame bytes.

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 Cellular NB-IoT device (NBIoTDevice)
  – 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 de-
scribing 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>
<tr>
<td>NBIoTDevice</td>
<td>Cellular NB-IoT 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:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoteXBeeDevice</td>
<td>Generic object, protocol independent</td>
</tr>
<tr>
<td>RemoteZigBeeDevice</td>
<td>ZigBee protocol</td>
</tr>
<tr>
<td>RemoteRaw802Device</td>
<td>802.15.4 protocol</td>
</tr>
<tr>
<td>RemoteDigiMeshDevice</td>
<td>DigiMesh protocol</td>
</tr>
<tr>
<td>RemoteDigiPointDevice</td>
<td>Point-to-multipoint protocol</td>
</tr>
</tbody>
</table>

**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`.
  - 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.

### 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

[...]

# 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.
node_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

[...]
# 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.

---

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.

---

```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()
```

```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()
```
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.

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.

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>
</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.

Refresh cached parameters

[...]

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

# Refresh the cached values.
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.

<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

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

(continues on next page)
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()</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**
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 `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`

### 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

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 \texttt{API\_MODE} or \texttt{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}.

<table>
<thead>
<tr>
<th>Example: Set and get parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
</tr>
</tbody>
</table>

### 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 \texttt{execute\_command()} method provided by all the XBee device classes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{execute_command(String)}</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

```python
[..., 

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

# Run the apply changes command.
local_xbee.execute_command("AC")

[...]
```

The \texttt{execute\_command()} method may fail for the following reasons:

- ACK of the command sent is not received in the configured timeout, throwing a \texttt{TimeoutException}.
- \textbf{Other errors caught as XBeeException:}
  - The operating mode of the device is not \texttt{API\_MODE} or \texttt{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}.

#### 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.
### Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enable_apply_changes(B)(enable_apply_changes(Boolean))</code></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><code>is_apply_changes_enabled()</code></td>
<td>Whether the XBee device is configured to apply parameter changes when they are set.</td>
<td></td>
</tr>
<tr>
<td><code>apply_changes()</code></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()`.

### Write configuration changes

```python
# 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><code>set_ip_addressing_mode()</code></td>
<td>Sets the IP addressing mode of the Wi-Fi module. The mode selected depends on the provided parameter.</td>
</tr>
<tr>
<td></td>
<td>• <strong>DHCP</strong>: Network settings are assigned by a server.</td>
</tr>
<tr>
<td></td>
<td>• <strong>STATIC</strong>: Network settings must be provided manually one by one.</td>
</tr>
</tbody>
</table>

**Configure IP addressing mode**
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

(continues on next page)
# Display the IP network settings that were assigned by the DHCP server.

```
print("- IP address: \$s" % local_xbee.get_ip_address())
print("- Subnet mask: \$s" % local_xbee.get_mask_address())
print("- Gateway IP address: \$s" % local_xbee.get_gateway_address())
print("- DNS IP address: \$s" % local_xbee.get_dns_address())

[...]
```

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:

- `_configBluetoothEnableDisable`
- `_configBluetoothConfigurePassword`
- `_configBluetoothReadMacAddress`

### 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()
```

(continues on next page)
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><code>update_bluetooth_password(String)</code></td>
<td>Specifies the new Bluetooth password of the XBee device.</td>
</tr>
</tbody>
</table>

#### Configuring or changing the Bluetooth password

```python
# 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. 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

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.

<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>
<tr>
<td><code>set_discovery_options(Set&lt;DiscoveryOptions&gt;)</code></td>
<td>Configures the discovery options (NO parameter) with the set of options. The set of discovery options contains the different <code>DiscoveryOptions</code> configuration values that are applied to the local XBee module when performing the discovery process. These options are the following: <code>DiscoveryOptions.APPEND_DD</code>: 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. <code>DiscoveryOptions.DISCOVER_MYSELF</code>: The local XBee device is returned as a discovered device. This option is valid for all protocols. <code>DiscoveryOptions.APPEND_RSSI</code>: 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.</td>
</tr>
</tbody>
</table>

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)
```

(continues on next page)
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>start_discovery_process()</td>
<td>Starts the discovery process, saving the remote XBee devices found inside the XBeeNetwork 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.

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):
    [...]  

# 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(...)  

# Define the discovery process finished callback.
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>discover_device(String)</td>
<td>Specify the node identifier of the XBee device to be found. Returns the remote XBee device whose node identifier equals the one provided or None if the device was not found. In the case of finding more than one device, it returns the first one.</td>
</tr>
<tr>
<td>discover_devices([String])</td>
<td>Specify the node identifiers of the XBee devices to be found. Returns a list with the remote XBee devices whose node identifiers equal those provided.</td>
</tr>
</tbody>
</table>

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

Discover specific devices

```python
# 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:
### Access discovered devices

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

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

# Discover a device based on a 64-bit address.
spec_device = xnet.get_device_by_64(x64addr)
if spec_device is None:
    print("Device with 64-bit addr: %s not found" % str(x64addr))

# Discover a device based on a Node ID.
spec_device = xnet.get_device_by_node_id(node_id)
if spec_device is not None:
    print("Device with node id: %s not found" % node_id)
```

#### 2.5.5.4 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>add_remote(RemoteXBeeDevice)</td>
<td>Specifies the remote XBee device to be added to the list of remote devices of the XBeeNetwork object.</td>
</tr>
<tr>
<td><strong>Notice</strong></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><strong>Note</strong></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.</td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
</tbody>
</table>

| add_remotes([RemoteXBeeDevice]) | Specifies the remote XBee devices to be added to the list of remote devices of the XBeeNetwork object. |
| **Notice** | that this operation does not join the remote XBee devices to the network; it just tells the network that it contains those devices. However, the devices have only been added to the device list, and may not be physically in the same network. |

Add a remote device manually to 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)  
[...]```

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.

```python
2.5. API reference
```
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.</td>
</tr>
</tbody>
</table>

**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.

Clear the list of remote 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 Bytestring, 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><strong>send_data(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)</strong></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 XBee16BitAddress.UNKNOWN_ADDRESS.</td>
</tr>
<tr>
<td>Raw802Device</td>
<td><strong>send_data(XBee16BitAddress, String or Bytearray, Integer)</strong></td>
<td>Specifies the 16-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td>DigiMeshDevice</td>
<td><strong>send_data(XBee64BitAddress, String or Bytearray, Integer)</strong></td>
<td>Specifies the 64-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td>DigiPointDevice</td>
<td><strong>send_data(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)</strong></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 XBee16BitAddress.UNKNOWN_ADDRESS.</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 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.

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 XBee device class.

### Get/set the timeout for synchronous operations

```python
[...]

NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 # 5 seconds

(continues on next page)```
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)
[...]

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:
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 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>send_data_async(RemoteXBeeDevice, String or Bytearray, 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 XBeeDevice object:

<table>
<thead>
<tr>
<th>XBee class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zig-BeeDevice</td>
<td>send_data_async(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)</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 XBee16BitAddress.UNKNOWN_ADDRESS.</td>
</tr>
<tr>
<td>Raw802.15.4Device</td>
<td>send_data_async(XBee16BitAddress, String or Bytearray, Integer)</td>
<td>Specifies the 16-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td>DigiMeshDevice</td>
<td>send_data_async(XBee64BitAddress, String or Bytearray, Integer)</td>
<td>Specifies the 64-bit destination address, the data to send and optionally the transmit options.</td>
</tr>
<tr>
<td>DigiPointDevice</td>
<td>send_data_async(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)</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 XBee16BitAddress.UNKNOWN_ADDRESS.</td>
</tr>
</tbody>
</table>

Send data asynchronously
# 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_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 XBee 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:

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

### Send broadcast data

```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:

- 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: examples/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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_data()</code></td>
<td>Specifies 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.</td>
</tr>
</tbody>
</table>

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

```
[...]
xb_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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_data_from(RemoteXBeeDevice, Integer)</td>
<td>Specifies the remote 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.</td>
</tr>
</tbody>
</table>

Read data from a specific remote XBee device (polling)

```
[...]

# 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)
[...]
```

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

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 XBee device class.

Example: Receive data with polling

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

```
examples/communication/ReceiveDataPollingSample
```
Data reception callback

This mechanism for reading data does not block your application. Instead, you can be notified when new data has been received if you are subscribed or registered to the data reception service using the add_data_received_callback method with a data reception callback as parameter.

Register for data reception

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

# Define callback.
def my_data_received_callback(xbee_message):
    address = xbee_message.remote_device.get_64bit_addr()
    data = xbee_message.data.decode("utf8")
    print("Received data from %s: %s" % (address, data))

# Add the callback.
device.add_data_received_callback(my_data_received_callback)

# Delete the callback
device.del_data_received_callback(my_data_received_callback)
```

When new data is received, your callback is executed providing as parameter an XBeeMessage object which contains the data and other useful 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.

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

Deregister data reception

```python
# 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 XBee 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 XBee 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><code>send_expl_data(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>

Every protocol-specific XBee device object with support for explicit data includes at least one more method to transmit unicast explicit data synchronously:
### 2.5. API reference

<table>
<thead>
<tr>
<th>XBeed Class</th>
<th>Method Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZigBeeDevice</td>
<td>send_expl_data(XBee64BitAddress, XBee16BitAddress, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</td>
<td>Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options. If the 16-bit address is unknown, use the XBee16BitAddress. UNKNOWN_ADDRESS.</td>
</tr>
<tr>
<td>DigiMeshDevice</td>
<td>send_expl_data(XBee64BitAddress, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</td>
<td>Specifies the 64-bit destination address, 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>
<tr>
<td>DigiPointDevice</td>
<td>send_expl_data(XBee64BitAddress, XBee16BitAddress, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</td>
<td>Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options. If the 16-bit address is unknown, use the XBee16BitAddress. UNKNOWN_ADDRESS.</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 XBee device class.

**Example: Transmit explicit synchronous unicast data**

The XBee 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:

```
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>send_expl_data_async(RemoteXBeeDevice, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</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>

Every protocol-specific XBee device object that supports explicit data includes at least one additional method to transmit unicast explicit data asynchronously:

<table>
<thead>
<tr>
<th>XBee class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZigBeeDevice</td>
<td>send_expl_data_async(XBee64BitAddress, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</td>
<td>Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options. If the 16-bit address is unknown, use the XBee16BitAddress. UNKNOWN_ADDRESS.</td>
</tr>
<tr>
<td>DigiMeshDevice</td>
<td>send_expl_data_async(XBee64BitAddress, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</td>
<td>Specifies the 64-bit destination address, 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>
<tr>
<td>DigiPointDevice</td>
<td>send_expl_data_async(XBee64BitAddress, XBee16BitAddress, Integer, Integer, Integer, Integer, String or Bytearray, Integer)</td>
<td>Specifies the 64-bit and 16-bit destination addresses in addition to the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options. If the 16-bit address is unknown, use the XBee16BitAddress. UNKNOWN_ADDRESS.</td>
</tr>
</tbody>
</table>

Send unicast explicit data asynchronously

```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 asynchronously using the remote object.
device.send_expl_data_async(remote_device, 0xA0, 0xA1, 0x1554, 0xC105, "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 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, 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**

```python
# 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` method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_api_output_mode()</code></td>
<td>Returns the API output mode of the data received by the XBee device.</td>
</tr>
<tr>
<td><code>set_api_output_mode(APIOutputMode)</code></td>
<td>Specifies the API output mode of the data received by the XBee device.</td>
</tr>
<tr>
<td></td>
<td>• <code>APIOutputMode.NATIVE</code>: The data received by the device will be output as</td>
</tr>
<tr>
<td></td>
<td>standard received data and it must be read using standard data-reading</td>
</tr>
<tr>
<td></td>
<td>methods. It does not matter if the data sent by the remote device was</td>
</tr>
<tr>
<td></td>
<td>sent in standard or explicit format.</td>
</tr>
<tr>
<td></td>
<td>• <code>APIOutputMode.EXPLICIT</code>: The data received by the device will be output</td>
</tr>
<tr>
<td></td>
<td>as explicit received data and it must be read using explicit data-</td>
</tr>
<tr>
<td></td>
<td>reading methods. It does not matter if the data sent by the remote</td>
</tr>
<tr>
<td></td>
<td>device was sent in standard or explicit format.</td>
</tr>
<tr>
<td></td>
<td>• <code>APIOutputMode.EXPLICIT_ZDO_PASSTHRU</code>: The data received by the device</td>
</tr>
<tr>
<td></td>
<td>will be output as explicit received data, like the <code>APIOutputMode.EXPLICIT</code> option. In addition, this mode also outputs as explicit data ZigBee Device Object (ZDO) packets received by the XBee module through the serial interface.</td>
</tr>
</tbody>
</table>

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()</code></td>
<td>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()
```

(continues on next page)
# Read data.

```python
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)

(continues on next page)```
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

[...]

# 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: \$s" % (address, 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 un-
subscribe 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)

# Delete the callback
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 (APIOutputMode.EXPLICIT or
APIOutputMode.EXPLICIT_ZDO_PASSTHRU) and another device sends non-explicit data, you receive an ex-
plicit message whose application layer field values are:

• Source endpoint: 0xE8
• Destination endpoint: 0xE8
• Cluster ID: 0x0011
• Profile ID: 0xC10

When an XBee module receives explicit data with these values, the message notifies both data reception callbacks
(explicit and non-explicit) in case you have registered them. 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, NB-IoT, 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.

- **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, NBIODevice, 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>

**Note:** NB-IoT modules only support UDP transmissions, so make sure you use that protocol when calling the previous methods.

### 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!"
```

(continues on next page)
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`, `NBIODevice`, 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><code>send_ip_data_async(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>

**Note:** NB-IoT modules only support UDP transmissions, so make sure you use that protocol when calling the previous methods.

Send network data asynchronously
# 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_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</code></td>
<td>Starts listening for incoming IP transmissions in the provided port.</td>
</tr>
</tbody>
</table>

## Receive IP data

```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(int)</code></td>
<td>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)
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

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)

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.
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

```python
# Delete the IP data callback.
```

(continues on next page)
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><code>send_sms(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 synchronously**

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

...
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`.

### 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 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(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`.  

---

2.5. API reference  

65
● 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

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

# Define the callback.
def my_sms_callback(sms_message):
    print("Received SMS from \%s: \%s\" % (sms_message.phone_number, sms_message.data))

# Add the callback.
device.add_sms_callback(my_sms_callback)
```

When a new SMS message is received, your callback is executed providing an SMSSMessage object as 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

```python
# Delete the SMS callback.
device.del_sms_callback(my_sms_callback)
```
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><code>send_bluetooth_data(Bytesarray)</code></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
# Instantiates 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`. 

2.5. API reference
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:

text

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)
device.open()

# Define the callback.
def my_bluetooth_data_callback(data):
    print("Data received from the Bluetooth interface >> \"\" % data.decode("utf-8"))

# Add the callback.
device.add_bluetooth_data_received_callback(my_bluetooth_data_callback)
```

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.

**Deregsister Bluetooth data reception**

```python
# Instantiate an XBee device object.
device = [...] 

def my_bluetooth_data_callback(data):
    [...] 

device.add_bluetooth_data_received_callback(my_bluetooth_data_callback)

# Delete the Bluetooth data callback.
device.del_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(Bytearray)</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"))

# Add the callback.
device.add_micropython_data_received_callback(my_micropython_data_callback)
```

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 `del_micropython_data_received_callback` method to unsubscribe the already-registered listener.

Deregister MicroPython data reception

```python
# Delete the MicroPython data callback.
device.del_micropython_data_received_callback(my_micropython_data_callback)
```
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:
examples/communication/micropython/ReceiveMicroPythonDataSample

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 `add_modem_status_received_callback`.

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

```python
# Delete the modem status callback.
device.del_modem_status_received_callback(my_modem_status_callback)
```
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

2.5.7 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()
```

(continues on next page)
# 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**

[...]

# 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`.
- 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

[...]

# 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.

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()

# Get the duty cycle of the DIO10_PWM0 line (PWM output).
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: %s" % io_sample.get_digital_value(ioLine))
```

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 XBee 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
...
# 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"))

# 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
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
[...]
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))
[...]
```

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 \texttt{RemoteXBeeDevice} object.
• The time in which the IO sample was received as an \texttt{Float} (calculated with Python standard \texttt{time.time}()).

To stop receiving notifications of new IO samples, remove the added callback using the \texttt{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 \texttt{del_io_sample_received_callback} method will raise a \texttt{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:
\texttt{examples/io/IOSamplingSample/IOSamplingSample.py}

2.5.8 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 \texttt{devices} and \texttt{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

# Get the logger of the devices module.
dev_logger = logging.getLogger(digi.xbee.devices.__name__)

# Get the logger of the devices module providing the name.
dev_logger = logging.getLogger("digi.xbee.devices")
```

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)  
[...]  
```

### 2.5.8.1 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>
</table>
| `enable_logger(name, level=logging.DEBUG)` | Enables the logger.  
  - name: the name of the module whose logger you want to activate.  
  - level: default DEBUG. The level you want to see. |

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)

[...]

Note: For further information about the Python logging module, see the Python logging module official documentation or the Python logging cookbook.

2.5.9 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
- 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

2.5.9.1 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.*

## 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.*

### 2.5.9.2 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.

2.5.9.3 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/SendDataSample

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
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 XBe 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 XBe 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.*

2.5.9.4 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 Frequently Asked Questions (FAQs)

The FAQ section contains answers to general questions related to the XBee Python Library.

2.5.10.1 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.
2.5.10.2 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.

Note: Note In UNIX systems, the complete name of the serial port contains the /dev/ prefix.
2.5.10.3 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.

2.5.10.4 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.

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.

2.5.10.5 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
```

2.5.11 API reference

Following is API reference material on major parts of XBee Python library.

2.5.11.1 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**

`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’)`

`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)

**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.NI` = NI
- `ATStringCommand.KY` = KY
- `ATStringCommand.NK` = NK
- `ATStringCommand.ZU` = ZU
- `ATStringCommand.ZV` = ZV
- `ATStringCommand.CC` = CC

**command**

String. AT Command alias.

**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
- `SpecialByte.XOFF_BYTE` = 19

**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

2.5. API reference
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.XH9_001_TX` = (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_DXX_XXX` = (29, 'XBP08-Dxx-xxx 868MHz')
- `HardwareVersion.XBP24B` = (30, 'XB24B: Low cost ZB PRO and PLUS S2B')
- `HardwareVersion.XB24_WF` = (31, 'XB24-WF: XBee 802.11 (Redpine module)')
- `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, 'XB24C (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, 'XBP24D: S2D SMT Reg')
HardwareVersion.S2D_TH_PRO = (52, 'XBP24D: S2D TH PRO')
HardwareVersion.S2D_TH_REG = (53, 'XBP24D: 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.XBEE3_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_NBIO_T_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:

```python
OperatingMode.AT_MODE = (0, 'AT mode')
OperatingMode.API_MODE = (1, 'API mode')
OperatingMode.ESCAPED_API_MODE = (2, 'API mode with escaped characters')
OperatingMode.UNKNOWN = (99, 'Unknown')
```

code

Integer. The operating mode code.

description

The operating mode description.

Type String

class digi.xbee.models.mode.OperatingMode (code, description)

Bases: enum.Enum

Enumerates the different operating modes. The operating mode determines how a device will interact with the XBee series 2 module.

Inherited properties:

- **name** (String): the name (id) of this OperatingMode.
- **value** (String): the value of this OperatingMode.

Values:

```python
OperatingMode.AT_MODE = (0, 'AT mode')
OperatingMode.API_MODE = (1, 'API mode')
OperatingMode.ESCAPED_API_MODE = (2, 'API mode with escaped characters')
OperatingMode.UNKNOWN = (99, 'Unknown')
```

code

Integer. The API output mode code.

description

The API output 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:

```python
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.IPAddressingMode (code, description)

Bases: enum.Enum

Enumerates the different IP addressing modes.

Values:

```python
IPAddressingMode.DHCP = (0, 'DHCP')
IPAddressingMode.STATIC = (1, 'Static')
```

code

Integer. The API output mode code.

description

The API output mode description.

Type String

class digi.xbee.models.mode.IPAddressingMode (code, description)

Bases: enum.Enum

Enumerates the different IP addressing modes.
**code**

Integer. The IP addressing mode code.

**description**

String. The IP addressing mode description.

digi.xbee.models.address module

class digi.xbee.models.address.XBee16BitAddress(address)

    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: \((0[xX])?\{0-9a-fA-F\}\{1,4}\)

    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 has less than 1 byte or more than 2.

    **COORDINATOR_ADDRESS** = <digi.xbee.models.address.XBee16BitAddress object> (0000).

    Type 16-bit address reserved for the coordinator (value

    **BROADCAST_ADDRESS** = <digi.xbee.models.address.XBee16BitAddress object> (FFFF).

    Type 16-bit broadcast address (value

    **UNKNOWN_ADDRESS** = <digi.xbee.models.address.XBee16BitAddress object> (FFFE).

    Type 16-bit unknown address (value
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.

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 XBee 64-bit address as byte array.
    
    **Raise:** ValueError: if length of address is less than 1 or greater than 8.

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> 00000000000FFFF).

Type 64-bit broadcast address (value

UNKNOWN_ADDRESS = <digi.xbee.models.address.XrBe64BitAddress object>

Type 64-bit unknown address (value
classmethod from_hex_string(address)
Class constructor. Instantiates a new XBe64BitAddress object from the provided hex string.

Parameters address (String) – The XBee 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 XBe64BitAddress object from the provided bytes.

Parameters args (8 Integers) – 8 integers that represent the bytes 1 to 8 of this XBe64BitAddress.

Raises ValueError – if the amount of arguments is not 8 or if any of the arguments is not between 0 and 255.

address
Bytarray. Bytarray representation of this XBe64BitAddress.

class digi.xbee.models.address.XrBeIMEIAddress(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 XBeIMEIAddress object with the provided parameters.

Parameters address (Bytesarray) – The XBee IMEI address as byte array.

Raises
• ValueError – if address is None.
• ValueError – if length of address greater than 8.

classmethod from_string(address)
Class constructor. Instantiates a new XBeIMEIAddress 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)$.

address
String. String representation of this XBeIMEIAddress.

digi.xbee.models.message module
class digi.xbee.models.message.XrBeMessage(data, remote_device, timestamp, broadcast=False)
Bases: object
This class represents a XBee message, which is formed by a \texttt{RemoteXBeeDevice} (the sender) and some data (the data sent) as a bytearray.

Class constructor.

\textbf{Parameters}

- \texttt{data (Bytearray)} – the data sent.
- \texttt{remote\_device (RemoteXBeeDevice)} – the sender.
- \texttt{broadcast (Boolean, optional, default=`False`) – flag indicating whether the message is broadcast (True) or not (False). Optional.}
- \texttt{timestamp} – instant of time when the message was received.

\texttt{to\_dict()}

Returns the message information as a dictionary.

\texttt{data}

Bytearray. Bytearray containing the data of the message.

\texttt{remote\_device}

\texttt{RemoteXBeeDevice}. The device that has sent the message.

\texttt{is\_broadcast}

Boolean. True to indicate that the message is broadcast, False otherwise.

\texttt{timestamp}

Integer. Instant of time when the message was received.

\texttt{class \texttt{digi.xbee.models.message.ExplicitXBeeMessage}(data, remote\_device, timestamp, source\_endpoint, dest\_endpoint, cluster\_id, profile\_id, broadcast=False)}

\texttt{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.

\textbf{Parameters}

- \texttt{data (Bytearray)} – the data sent.
- \texttt{remote\_device (RemoteXBeeDevice)} – the sender device.
- \texttt{timestamp} – instant of time when the message was received.
- \texttt{source\_endpoint (Integer)} – source endpoint of the message. 1 byte.
- \texttt{dest\_endpoint (Integer)} – destination endpoint of the message. 1 byte.
- \texttt{cluster\_id (Integer)} – cluster id of the message. 2 bytes.
- \texttt{profile\_id (Integer)} – profile id of the message. 2 bytes.
- \texttt{broadcast (Boolean, optional, default=`False`) – flag indicating whether the message is broadcast (True) or not (False). Optional.}

\texttt{to\_dict()}

Returns the message information as a dictionary.

\texttt{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
Bytearray. Bytearray containing the data of the message.

class digi.xbee.models.message.SMSSMessage(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 SMSSMessage 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.

```python
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:
**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.

```python
class digi.xbee.models.options.XBeeLocalInterface(code, description)

Bases: enum.Enum
```
Enumerates the different interfaces for the `UserDataRelayPacket` 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.
digi.xbee.models.protocol module

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, ‘TCP SSL’)

**code**
  IP protocol code.
  
  **Type** Integer

**description**
  IP protocol description.
  
  **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’)

**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.SOCKE_T_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.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

2.5. API reference 115
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.UNKNOW = (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’)

116 Chapter 2. Contents
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.NOJOINABLE_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).’)

AssociationIndicationStatusCOORDINATOR_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, de-
scription)

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_LOW_POWER = (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’)
    DeviceCloudStatusUNKNOWN_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’)

code
Integer. The network discovery status code.
**description**

String. The network discovery status description.

**digi.xbee.packets package**

**Submodules**

**digi.xbee.packets.aft module**

```python
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.SEND_DATA_REQUEST = (40, ‘Send Data Request’)
ApiFrameType.DEVICE_RESPONSE = (42, ‘Device Response’)
ApiFrameType.USER_DATA_RELAY_REQUEST = (45, ‘User Data Relay Request’)
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 Request (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.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.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.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.
   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 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 XBee packet created.

Return type   XBeePacket

Raises InvalidPacketException – if something is wrong with raw and the packet can-
not 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 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.base.GenericXBeePacket(rf_data)
  Bases: digi.xbee.packets.base.XBeeAPIPacket

  This class represents a basic and Generic XBee 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.

  See also:

  factory
  XBeeAPIPacket

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:
**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.base.UnknownXBeePacket(api_frame, rf_data)**

Based on **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**

**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.

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.

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.

Return type the frame type of this packet.
Return type \texttt{ApiFrameType}

See also:

\texttt{ApiFrameType}

\texttt{get\_frame\_type\_value()}\n
Returns the frame type integer value of this packet.

\textbf{Return type} Integer

See also:

\texttt{ApiFrameType}

\texttt{is\_broadcast()}\n
Returns whether this packet is broadcast or not.

\textbf{Returns} True if this packet is broadcast, False otherwise.

\textbf{Return type} Boolean

\texttt{needs\_id()}\n
Override method.

See also:

\texttt{XBeeAPIPacket.needs\_id()}

\texttt{output(escaped=False)}\n
Returns the raw bytearray of this XBeePacket, ready to be send by the serial port.

\textbf{Parameters} \texttt{escaped(Boolean)} – indicates if the raw bytearray will be escaped or not.

\textbf{Returns} raw bytearray of the XBeePacket.

\textbf{Return type} Bytearray

\texttt{to\_dict()}\n
Returns a dictionary with all information of the XBeePacket fields.

\textbf{Returns} dictionary with all information of the XBeePacket fields.

\textbf{Return type} Dictionary

\texttt{static unescape\_data(data)}\n
Un-escapes the provided bytearray data.

\textbf{Parameters} \texttt{data(Bytearray)} – the bytearray to unescape.

\textbf{Returns} data unescaped.

\textbf{Return type} Bytearray
digi.xbee.packets.cellular module

digi.xbee.packets.cellular.PATTERN_PHONE_NUMBER = r'^\+?\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.XBeeAPIPacket

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
XBeeAPIPacket

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 Bytestring

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:
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
    XBeesAPIPacket

    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()
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:**

`XBeePacket.create_packet()`

`XBeeAPIPacket._check_api_packet()`

**needs_id**()

Override method.

**See also:**

`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:

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 \(\text{escaped}=False\)

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

2.5. API reference 139
**static create_packet**(raw, operating_mode)

Override method.

Returns `ATCommQueuePacket`

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_QUEUE`.
- `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()`

**command**

String. AT command.

**parameter**

Bytarray. 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:
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.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) – 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
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.

• **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()
```

**command**
String. AT command.

**command_value**
Bytearray. AT command value.

**status**
ATCommandStatus. 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.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()

x64bit_source_addr
XBee64BitAddress. 64-bit source address.

x16bit_source_addr
XBee16BitAddress. 16-bit source address.

receive_options
Integer. Receive options bitfield.

rf_data
Bytearray. 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:

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.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:

`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.

`command`
String. AT command.

`parameter`
Bytarray. 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.

**Returns**  checksum value of this XBeePacket.

**Return type**  Integer

See also:

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) – 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.
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.

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:
 XBee Python Library Documentation, Release 1.2.0

**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
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:

TransmitOptions
XBee16BitAddress
XBee64BitAddress
XBeeAPIPacket

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
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`

```python
class digi.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: digi.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:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

```python
needs_id()
```
Override method.

See also:

```python
XBeeAPIPacket.needs_id()
```

```python
x16bit_dest_addr
```
`XBee16BitAddress`. 16-bit destination address.

```python
transmit_retry_count
```
Integer. Transmit retry count value.

```python
transmit_status
```
`TransmitStatus`. Transmit status.

```python
discovery_status
```
`DiscoveryStatus`. Discovery status.

```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:
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 check-
      sum 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
    Bytestring. 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.
    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 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. 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
 XBee64BitAddress
 TransmitOptions
 XBeeAPIPacket

**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** *

`XBee64BitAddress`. 64-bit destination address.

**x16bit_dest_addr** *

`XBee16BitAddress`. 16-bit destination address.

**transmit_options** *

Integer. Transmit options bitfield.

**broadcast_radius** *

Integer. Broadcast radius.
source_endpoint
   Integer. Source endpoint of the transmission.

dest_endpoint
   Integer. Destination endpoint of the transmission.

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

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  

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 `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()`

`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`  
Bytearray. Received RF data.
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** *(Bytearray, 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

- **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).
- **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_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()
```

**request_id**
Integer. Request ID of the packet.

**transport**
Integer. Transport (reserved).

**flags**
Integer. Flags (reserved).

**target**
String. Request target of the packet.

**request_data**
Bytearray. Data of the device request.

**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.DeviceResponsePacket(frame_id, request_id, response_data=None)

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:

`XBeePacket.create_packet()`
`XBeeAPIPacket._check_api_packet()`

`needs_id()`
Override method.
See also:

`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:

`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.

    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:

```python
XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()
```

`needs_id()`
Override method.
See also:

```python
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:

```python
factory
```

`get_frame_spec_data()`
Override method.

See also:

2.5. API reference 183
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.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

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:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

path
String. Path of the file to upload to Device Cloud.

content_type
String. The content type of the file to upload.

options
SendDataRequestOptions. File upload operation options.

file_data
Bytearray. Data of the file to upload.

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.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:**

`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广播()

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.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 checksum 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
   Bytearray. 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.

      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**  Bytestring

**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**  Bytestring

**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)*

**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:**
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:
XBee Python Library Documentation, Release 1.2.0

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

- 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 check-
  sum 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

g_get_frame_spec_data()
Override method.
See also:

XBeePacket.get_frame_spec_data()

g_get_frame_type()
Returns the frame type of this packet.

Returns: the frame type of this packet.
Return type: ApiFrameType
See also:

ApiFrameType

g_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).

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()

transmit_status
TransmitStatus. Transmit 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:
**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.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()`

`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.

`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.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()

xl6bit_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.

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.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:

- ReceiveOptions
- XBee64BitAddress
- XBeeAPIPacket

**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()

**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

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.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 (XBeel16BitAddress)` – 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
```

```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 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()
needs_id()
    Override method.

    See also:

        XBeeAPIPacket.needs_id()

xl6bit_source_addr
    XBee16BitAddress. 16-bit source address.

rssi
    Integer. Received Signal Strength Indicator (RSSI) value.

receive_options
    Integer. Receive options bitfield.

rf_data
    Bytestring. 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 (Bytestring, 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*.

• *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()
```

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  Bytestring

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 (Bytestring) – the bytearray to unescape.

Returns data unescaped.

Return type  Bytestring

src_interface
XBeeLocalInterface. Source local interface.
data
  Bytearray. Received data.

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.

See also:

IOSample
  ipaddress.IPv4Address
  ReceiveOptions
  XBeeAPIPacket

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
static create_packet (raw, operating_mode)

Override method.

Returns RemoteATCommandWifiPacket

Raises

- **InvalidPacketException** – if the Bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + frame id + dest. addr. (8 bytes) + transmit options + command (2 bytes) + 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_REQUEST_WIFI.

- **InvalidOperatingModeException** – if operating_mode is not supported.

See also:

XBeepacket.create_packet()
XBeeAPacket._check_api_packet()

needs id()

Override method.

See also:

XBeeAPacket.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 di.ige.xbee.packets.wifi.RemoteATCommandResponseWifiPacket(frame_id, source_address, command, response_status, comm_value=None)

Bases: di.ige.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:

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

source_address
ipaddress.IPv4Address. IPv4 source address.

custom
String. AT command.

status
ATCommandStatus. AT command response status.

command_value
Bytearray. AT command value.

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.
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'>
```

```python
>>> 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

```python
>>> a=0xFFF
>>> print([i for i in int_to_bytes(a)])
[255, 254]
>>> print(type(int_to_bytes(a)))
<type 'bytearray'>
```
digixbee.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
```

digixbee.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 0xA0A0A
>>> print(bytes_to_int(x))
657930
>>> b = bytearray([0x0A, 0xAA])  #this is 0xAAA
>>> print(bytes_to_int(b))
2730
```

digixbee.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.
#0x31,0x30,0x30 <== 1,0,0
>>> print(ascii_to_int(x))
100
```

digixbee.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.

2.5. API reference
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**:

- **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
```

```python
>>> a = 8
>>> print(hex_to_string(int_to_length(a)))
00 08
```

```python
>>> a = 200
>>> print(hex_to_string(int_to_length(a)))
00 CB
```

```python
>>> a = 0xFF00
>>> print(hex_to_string(int_to_length(a)))
FF 00
```

```python
>>> 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 given name.

Parameters `name` (String) – the name of the logger to disable.

Submodules

digi.xbee.devices module

class digi.xbee.devices.AbstractXBeeDevice(local_xbee_device=None, serial_port=None, sync_ops_timeout=4)

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.

See also:

XBeeDevice
XBeeSerialPort

2.5. API reference
LOG_PATTERN = '{port:<6s}{event:<12s}{opmode:<20s}{content:<50s}'

Pattern used to log packet events.

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.

get_parameter (parameter)

Returns the value of the provided parameter via an AT Command.

Parameters parameter (String) – parameter to get.

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.

• InvalidOperationException – if the XBee device’s operating mode is 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_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 XBees device’s serial port is closed.

• InvalidOperationException – if the XBee device’s operating mode is 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.
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.

`apply_changes()`

Applies changes via `AC` command.

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.

`write_changes()`

Writes configurable parameter values to the non-volatile memory of the XBees 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 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.

`reset()`

Performs a software reset on this XBees device and blocks until the process is completed.

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.

`read_device_info()`

Updates all instance parameters reading them from the XBees 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.

**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

**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.

**get_hardware_version()**
Returns the hardware version of the XBee device.

**Returns** the hardware version of the XBee device.

**Return type** 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

**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 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.

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_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.

:raises All exceptions raised by XBeeDevice.set_parameter().:

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**

- 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_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**

*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.

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 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`

**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`

**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**()

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

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

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 XBee device’s serial port is closed.
• `InvalidOperatingModeException` – 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_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.

`log`
Logger. The XBee device logger.

`class`  `digi.xbee.devices.XBeeDevice`  (`port`, `baud_rate`, `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`)

`Bases: digi.xbee.devices.AbstractXBeeDevice`

This class represents a non-remote generic XBee device.

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.

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.
• `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=serial.PARITY_NONE): comm port parity.
• `flow_control` (*FlowControl.NONE: None): comm port flow control.
• `_sync_ops_timeout` (*integer, default=3): comm port read timeout.

Raises  All exceptions raised by PySerial’s Serial class constructor.
See also:

PySerial documentation: http://pyserial.sourceforge.net

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 XBe device created.

   Return type  XBeeDevice

   Raises  SerialException – if the port you want to open does not exist or is already opened.

   See also:

   XBeeDevice

open()
   Opens the communication with the XBe device and loads some information about it.

   Raises
   • TimeoutException – if there is any problem with the communication.
   • 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.
   • XBeeException – if the XBe device is already open.

close()
   Closes the communication with the XBe device.

   This method guarantees that all threads running are stopped and the serial port is closed.

get_parameter(param)
   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.

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.
- XBeeException – if the XBee device’s serial port is closed.
- XBeeException – if the status of the response received is not OK.

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.

- **XBeeException** – if the XBee device’s serial port is closed.

- **XBeeException** – if the status of the response received is not OK.

**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()
See also:

```
XBeeDevice.has_packets()
```

**flush_queues()**
Flushes the packets queue.

**reset()**
Override method.

See also:

```
AbstractXBeeDevice.reset()
```

**add_packet_received_callback(callback)**
Override.

**add_data_received_callback(callback)**
Override.

**add_modem_status_received_callback(callback)**
Override.

**add_io_sample_received_callback(callback)**
Override.

**add_expl_data_received_callback(callback)**
Override.

**add_user_data_relay_received_callback(callback)**
Override.

**add_bluetooth_data_received_callback(callback)**
Override.

**add_micropython_data_received_callback(callback)**
Override.

**del_packet_received_callback(callback)**
Override.

**del_data_received_callback(callback)**
Override.

**del_modem_status_received_callback(callback)**
Override.

**del_io_sample_received_callback(callback)**
Override.

**del_expl_data_received_callback(callback)**
Override.

**del_user_data_relay_received_callback(callback)**
Override.

**del_bluetooth_data_received_callback(callback)**
Override.
del_micropython_data_received_callback(callback)
Override.

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)
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

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 XBeé device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeé device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

disable_bluetooth()
Disables the Bluetooth interface of this XBeé 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 XBeé device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeé 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 XBeé 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 XBeé device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeé 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.

Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBeé device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeé device’s operating mode is 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 XBeé device.

Returns the 16-bit address of the XBeé device.
Return type `XBee6BitAddress`

See also:

`XBee6BitAddress`

`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**: `int`

- **Raises**
  - `TimeoutException` – if the response is not received before the read timeout expires.
  - `XBeeException` – if the XBee device’s serial port is closed.
  - `InvalidOperatingModeException` – if the XBee device’s operating mode is 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()`

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`
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_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_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

read_device_info()
Updates all instance parameters reading them from 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.

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**

**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)

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_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.

Raises All exceptions raised by **XBeeDevice.set_parameter()**.

**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)

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:**

2.5. API reference  
263
**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.
**write_changes()**

Writes configurable parameter values to the non-volatile memory of the XBe 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 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.

**class digi.xbee.devices.Raw802Device**(port, baud_rate)

Bases: `digi.xbee.devices.XBeeDevice`

This class represents a local 802.15.4 XBe device.

Class constructor. Instantiates a new `Raw802Device` with the provided parameters.

**Parameters**

- **port** (Integer or String) – serial port identifier. Integer: number of XBe 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.

:raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

```
XBeeDevice
XBeeDevice.__init__()
```

**open()**

Override.

**Raises** `XBeeException` – if the protocol is invalid.

:raises All exceptions raised by `XBeeDevice.open()`.

See also:

```
XBeeDevice.open()
```
get_network()  
Override.  
See also:  

XBeeDevice.get_network()  

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)
   Override.

add_data_received_callback (callback)
   Override.

add_expl_data_received_callback (callback)
   Override.

add_io_sample_received_callback (callback)
   Override.

add_micropython_data_received_callback (callback)
   Override.

add_modem_status_received_callback (callback)
   Override.

add_packet_received_callback (callback)
   Override.

add_user_data_relay_received_callback (callback)
   Override.

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.

close ()
   Closes the communication with the XBee device.

   This method guarantees that all threads running are stopped and the serial port is closed.

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)
Override.

del_data_received_callback (callback)
Override.

del_expl_data_received_callback (callback)
Override.

del_io_sample_received_callback (callback)
Override.

del_micropython_data_received_callback (callback)
Override.

del_modem_status_received_callback (callback)
Override.

del_packet_received_callback (callback)
Override.

del_user_data_relay_received_callback (callback)
Override.

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 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.

**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.

**flush_queues**()

Flushes the packets queue.

**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 XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is 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()

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_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_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)
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_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.

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 re-
    ceive 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()

Updates all instance parameters reading them from 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.

### 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:

```python
AbstractXBeeDevice.reset()
```
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:
XBeedevicessend_micropython_data()
XBeedevicessend_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 XBeedevicess_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 XBeedevicess_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:
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.
- **XBeeException** – if the XBee device’s serial port is closed.
- **XBeeException** – if the status of the response received is not OK.

**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:

AbstractXBeeDevice._send_packet()

**send_packet_sync_and_get_response** *(packet_to_send)*
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:

**serial_port**
Returns the serial port associated to the XBee device.

**Returns** the serial port associated to the XBee device.

**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*)
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_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.

:raises All exceptions raised by *XBeeDevice.set_parameter()*.

**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*)
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** (*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 **
XBee Python Library Documentation, Release 1.2.0

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is 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.

```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.

```python
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.DigiMeshDevice(port, baud_rate)
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** (*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.

:raises All exceptions raised by XBeeDevice.__init__() constructor.

See also:

XBeeDevice
XBeeDevice.__init__() 

open()
Override.

    Raises **XBeeException** – if the protocol is invalid.

    :raises All exceptions raised by XBeeDevice.open().

See also:

XBeeDevice.open()

get_network()
Override.

See also:

XBeeDevice.get_network()

get_protocol()
Override.

See also:

XBeeDevice.get_protocol()
**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()`

**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)*

Override.

**add_data_received_callback** *(callback)*

Override.

**add_expl_data_received_callback** *(callback)*

Override.

**add_io_sample_received_callback** *(callback)*

Override.

**add_micropython_data_received_callback** *(callback)*

Override.

**add_modem_status_received_callback** *(callback)*

Override.

**add_packet_received_callback** *(callback)*

Override.

**add_user_data_relay_received_callback** *(callback)*

Override.

**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.

**close** *

Closes the communication with the XBee device.

This method guarantees that all threads running are stopped and the serial port is closed.

**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)
 Override.

del_data_received_callback (callback)
 Override.

del_expl_data_received_callback (callback)
 Override.

del_io_sample_received_callback (callback)
 Override.

del_micropython_data_received_callback (callback)
 Override.

del_modem_status_received_callback (callback)
 Override.

del_packet_received_callback (callback)
 Override.

del_user_data_relay_received_callback (callback)
 Override.

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.

**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.

**flush_queues** *
Flushes the packets queue.

**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()**

Returns the API output mode of the XBe device.

The API output mode determines the format that the received data is output through the serial interface of the XBe device.

**Returns**  
The API output mode of the XBe device.

**Return type**  
`APIOutputMode`

**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:

**APIOutputMode**

---

**get_bluetooth_mac_addr()**

Reads and returns the EUI-48 Bluetooth MAC address of this XBe 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  Bytarray

**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_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)
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_sync_ops_timeout()
Returns the serial port read timeout.

Returns the serial port read timeout in seconds.
Return type Integer

get_xbee_device_callback()
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.

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 re-
   ceive 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()

Updates all instance parameters reading them from 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.

### 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()
```

---

2.5. API reference
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.
• XBeeException – if the XBee device’s serial port is closed.
• XBeeException – if the status of the response received is not OK.

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.
- `XBeeException` – if the XBee device’s serial port is closed.
- `XBeeException` – if the status of the response received is not OK.

**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)
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.

Returns the serial port associated to the XBee device.

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)
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_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.

:raises All exceptions raised by XBeeDevice.set_parameter():

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

```
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

```
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*)

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** (*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.

**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.

``` 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.DigiPointDevice(port, baud_rate)
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.

:raises All exceptions raised by `XBeeDevice.__init__()` constructor:

See also:

`XBeeDevice`

`XBeeDevice.__init__()`

```python
open()
```

Override.

*Raises **XBeeException*** – if the protocol is invalid.

:raises All exceptions raised by `XBeeDevice.open()`:

See also:

`XBeeDevice.open()`

```python
get_network()
```

Override.

See also:

`XBeeDevice.get_network()`

```python
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)
Override.

add_data_received_callback (callback)
Override.

add_expl_data_received_callback (callback)
Override.

add_io_sample_received_callback (callback)
Override.

add_micropython_data_received_callback (callback)
Override.

add_modem_status_received_callback (callback)
Override.

add_packet_received_callback (callback)
Override.

add_user_data_relay_received_callback (callback)
Override.

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.

close()
Closes the communication with the XBee device.

This method guarantees that all threads running are stopped and the serial port is closed.

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)
Override.

del_data_received_callback(callback)
Override.

del_expl_data_received_callback(callback)
Override.

del_io_sample_received_callback(callback)
Override.

del_micropython_data_received_callback(callback)
Override.

del_modem_status_received_callback(callback)
Override.

del_packet_received_callback(callback)
Override.

del_user_data_relay_received_callback(callback)
Override.

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.

** 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.

** flush_queues () **

Flushes the packets queue.

** 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 XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is 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()

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_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  Bytestring

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_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)

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_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.

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()**

Updates all instance parameters reading them from 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.

**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:

```python
AbstractXBeeDevice.reset()
```
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.

• XBeeException – if the XBee device’s serial port is closed.

• XBeeException – if the status of the response received is not OK.

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.
- *XBeeException* – if the XBee device’s serial port is closed.
- *XBeeException* – if the status of the response received is not OK.

**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)
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.

Returns the serial port associated to the XBee device.

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)*
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_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.

:raises All exceptions raised by `XBeeDevice.set_parameter()`.

**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)
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(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.

`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.

`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, baud_rate)
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.

:raises All exceptions raised by *XBeeDevice.__init__()*. constructor.

See also:

*XBeeDevice*
*XBeeDevice.__init__()*

open ()
Override.

    Raises *XBeeException* – if the protocol is invalid.

:raises All exceptions raised by *XBeeDevice.open()*.

See also:

*XBeeDevice.open()*

get_network ()
Override.

See also:

*XBeeDevice.get_network()*

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()

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()

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.
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
XBeePacket

add_bluetooth_data_received_callback(*callback*)
Override.

add_data_received_callback(*callback*)
Override.

add_expl_data_received_callback(*callback*)
Override.

add_io_sample_received_callback(*callback*)
Override.
add_micropython_data_received_callback\(/callback\)
Override.

add_modem_status_received_callback\(/callback\)
Override.

add_packet_received_callback\(/callback\)
Override.

add_user_data_relay_received_callback\(/callback\)
Override.

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.

close()
Closes the communication with the XBee device.

This method guarantees that all threads running are stopped and the serial port is closed.

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\(callback\)
Override.
del_data_received_callback (callback)
Override.

del_expl_data_received_callback (callback)
Override.

del_io_sample_received_callback (callback)
Override.

del_micropython_data_received_callback (callback)
Override.

del_modem_status_received_callback (callback)
Override.

del_packet_received_callback (callback)
Override.

del_user_data_relay_received_callback (callback)
Override.

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.

 Raises

• TimeoutException – if the response is not received before the read timeout expires.
• XBeeException – if the XBee device’s serial port is closed.
InvalidOperatingModeException – if the XBee device’s operating mode is 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()

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_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_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)

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_sync_ops_timeout()

Returns the serial port read timeout.

**Returns**

- the serial port read timeout in seconds.

**Return type** Integer

get_xbee_devicecallbacks()

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.

**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:

```python
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()**

Updates all instance parameters reading them from 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.

ATTCommandException – 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()

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.
- **XBeeException** – if the XBee device’s serial port is closed.
- **XBeeException** – if the status of the response received is not OK.

**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 \((\text{data, transmit_options}=0)\)

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.
- `XBeeException` – if the XBee device’s serial port is closed.
- `XBeeException` – if the status of the response received is not OK.

send_micropython_data \((\text{data})\)

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 \((\text{packet, sync=False})\)

Override method.

See also:

- `AbstractXBeeDevice._send_packet()`

send_packet_sync_and_get_response \((\text{packet_to_send})\)

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.

    Returns the serial port associated to the XBee device.

    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)
    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_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.

  :raises All exceptions raised by `XBeeDevice.set_parameter()`:

**set_dio_change_detection** *(io_lines_set)*

Sets the digital IO lines to be monitored and sampled whenever their status changes.

- **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)`
Sets the operating PAN ID of the XBee device.

   Parameters **value** (*Bytes*) – 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(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.
**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_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.

**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 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.

**class**

digi.xbee.devices.IPDevice *(port, baud_rate)*

**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.

:raises All exceptions raised by XBeeDevice.__init__() constructor.

See also:
XBeeDevice
XBeeDevice.__init__()

read_device_info()
Override.

See also:

AbstractXBeeDevice.read_device_info()

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

```python
add_ip_data_received_callback(callback)

Adds a callback for the event IPDataReceived.

Parameters callback (Function) – the callback. Receives one argument.

• The data received as an IMessage

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.
```

```python
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.
```

```python
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.
```

```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. 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.
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** (*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.

`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.

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)
  Override.

add_io_sample_received_callback(callback)
  Override.

add_micropython_data_received_callback(callback)
  Override.

add_modem_status_received_callback(callback)
  Override.

add_packet_received_callback(callback)
  Override.
add_user_data_relay_received_callback(callback)
Override.

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.

close()
Closes the communication with the XBee device.

This method guarantees that all threads running are stopped and the serial port is closed.

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)
Override.

del_io_sample_received_callback(callback)
Override.

del_micropython_data_received_callback(callback)
Override.

del_modem_status_received_callback(callback)
Override.
del_packet_received_callback (callback)
   Override.

del_user_data_relay_received_callback (callback)
   Override.

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.
   Raises
      • TimeoutException – if the response is not received before the read timeout expires.
      • XBeeException – if the XBee device’s serial port is closed.
      • InvalidOperatingModeException – if the XBee device’s operating mode is 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:
XBee4BitAddress

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()

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_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)
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 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 the passed IOLINE has no PWM capability.

See also:

IOLine
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()

Opens the communication with the XBee device and loads some information about it.

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 XBee device is already open.

operating_mode

Returns this XBee device’s operating mode.

Returns OperatingMode. This XBee device’s 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()

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:

```python
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:

```python
XBeeDevice.send_bluetooth_data()
XBeeDevice.send_user_data_relay()
```

**send_packet**(*packet*, **sync=False**)  

Override method.

See also:

```python
AbstractXBeeDevice._send_packet()
```

**send_packet_sync_and_get_response**(*packet_to_send*)

Override method.

See also:

```python
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:

```python
XBeeLocalInterface
```

**serial_port**

Returns the serial port associated to the XBee device.

Returns the serial port associated to the XBee device.

Return type **XBeeSerialPort**

See also:

```python
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)**

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_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_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.

**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.

**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, baud_rate)`

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.

:raises All exceptions raised by `XBeeDevice.__init__()` constructor.: 

See also:

`XBeeDevice`

`XBeeDevice.__init__()`

`open()` 

Override.

Raises `XBeeException` – if the protocol is invalid.

:raises All exceptions raised by `XBeeDevice.open()`.

See also:

`XBeeDevice.open()`

`get_protocol()` 

Override.

See also:
XBeeDevice.get_protocol()

read_device_info()
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 SMSReceived.
- Parameters callback (Function) – the callback. Receives one argument.
  - The data received as an SMSMessage

del_sms_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.

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_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)
   Override.

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 IPDataReceived.
   Parameters callback(Function) – the callback. Receives one argument.
   • The data received as an IMessage

add_micropython_data_received_callback(callback)
   Override.

add_modem_status_received_callback(callback)
   Override.

add_packet_received_callback(callback)
   Override.

add_user_data_relay_received_callback(callback)
   Override.

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.

**close()**

Closes the communication with the XBee device.

This method guarantees that all threads running are stopped and the serial port is closed.

**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**(callback)

Override.

**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_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_micropython_data_received_callback \((\text{callback})\)
Override.

del_modem_status_received_callback \((\text{callback})\)
Override.

del_packet_received_callback \((\text{callback})\)
Override.

del_user_data_relay_received_callback \((\text{callback})\)
Override.

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 \((\text{value})\)
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 \((\text{parameter})\)
Executes the provided 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.

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()**

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_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

gdio_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:

```python
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:

```python
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:

```
ipaddress.IPv4Address
```

```python
def get_network()
    # Deprecated.
    # This protocol does not support the network functionality.
```

```python
def 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
```

```python
def get_pan_id()
    # Deprecated.
    # Operation not supported in this protocol. This method will raise an AttributeError.
```

```python
def get_parameter(param)
    # Override.
    # See also:
    # AbstractXBeeDevice.get_parameter()
```

```python
def 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_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.

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.

```python
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.

```python
reset()
```
Override method.

See also:

- `AbstractXBeeDevice.reset()`

```python
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`.  

---

2.5. API reference 373
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.
• OperationNotSupportedError – 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)`
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.

Returns the serial port associated to the XBee device.

Returns 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)
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_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_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:

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_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.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is 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:**
**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.

**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, baud_rate)
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.

:raises All exceptions raised by XBeeDevice.__init__() constructor.:  

See also:

```
XBeeDevice
XBeeDevice.__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.

```python
add_sms_callback (callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
del_sms_callback (callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
send_sms (phone_number, data)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
send_sms_async (phone_number, data)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
add_bluetooth_data_received_callback (callback)
```

Override.

```python
add_data_received_callback (callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
add_expl_data_received_callback (callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
add_io_sample_received_callback (callback)
```

Deprecated.

Operation not supported in this protocol. This method will raise an `AttributeError`.

```python
add_ip_data_received_callback (callback)
```

Adds a callback for the event `IPDataReceived`.

Parameters

**callback** (*Function*) – the callback. Receives one argument.

• The data received as an `IPMessage`

```python
add_micropython_data_received_callback (callback)
```

Override.

```python
add_modem_status_received_callback (callback)
```

Override.

```python
add_packet_received_callback (callback)
```

Override.

```python
add_user_data_relay_received_callback (callback)
```

Override.

```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.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

`close()`
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

`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)
Override.

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 *IPDataReceived* event.

**Parameters**

- **callback (Function)** – the callback to delete.

**Raises**

- **ValueError** – if *callback* is not in the callback list of *IPDataReceived* event.

**del_micropython_data_received_callback** *(callback)*
Override.

**del_modem_status_received_callback** *(callback)*
Override.

**del_packet_received_callback** *(callback)*
Override.

**del_user_data_relay_received_callback** *(callback)*
Override.

**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.

**Raises**

- **TimeoutException** – if the response is not received before the read timeout expires.
- **XBeeException** – if the XBee device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBee device’s operating mode is 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 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()
Returns the API output mode of the XBe device.
The API output mode determines the format that the received data is output through the serial interface of the XBe device.

Returns the API output mode of the XBe device.

Return type APIOutputMode

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:

`APIOutputMode`

`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_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.

`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 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 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()

  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)
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\_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()
   Override.
   Raises XBeeException – if the protocol is invalid.
   :raises All exceptions raised by XBeeDevice.open():
   See also:

XBeeDevice.open()

operating_mode
   Returns this XBee device’s operating mode.
   Returns OperatingMode. This XBee device’s operating mode.

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()
   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:
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 ()
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:

```python
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_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.

```python
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:

```python
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)*

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.

Returns the serial port associated to the XBee device.

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 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.

```python
set_api_output_mode(api_output_mode)
```
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`

```python
set_dest_address(addr)
```
Deprecated.

Operation not supported in this protocol. Use `IPDevice.set_dest_ip_addr()` instead. This method will raise an `AttributeError`.

```python
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 XBees related exception.

See also:

`ipaddress.IPv4Address`

```python
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.
- ATCommandException – if the response is not as expected.

See also:

IOLine
IOMode

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 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.

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, baud_rate)

Bases: digi.xbee.devices.LPWANDevice

This class represents a local NB-IoT 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.

:raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

```python
XBeeDevice
XBeeDevice.__init__()
```

**open()**

Override.

Raises `XBeeException` – if the protocol is invalid.

:raises All exceptions raised by `XBeeDevice.open()`.

See also:

```python
XBeeDevice.open()
```

**get_protocol()**

Override.

See also:

```python
XBeeDevice.get_protocol()
```

**add_bluetooth_data_received_callback** (*callback*)

Override.

**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 `IPDataReceived`.

Parameters **callback** (*Function*) – the callback. Receives one argument.

- The data received as an `IPMessage`
add_micropython_data_received_callback(callback)
Override.

add_modem_status_received_callback(callback)
Override.

add_packet_received_callback(callback)
Override.

add_sms_callback(callback)
Deprecated.
Operation not supported in this protocol. This method will raise an AttributeError.

add_user_data_relay_received_callback(callback)
Override.

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.

close()
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

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)
  Override.

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 IPDataReceived event.

  Parameters callback(Function) – the callback to delete.

  Raises ValueError – if callback is not in the callback list of IPDataReceived event.

del_micropython_data_received_callback(callback)
  Override.

del_modem_status_received_callback(callback)
  Override.

del_packet_received_callback(callback)
  Override.

del_sms_callback(callback)
  Deprecated.
  Operation not supported in this protocol. This method will raise an AttributeError.

del_user_data_relay_received_callback(callback)
  Override.

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.

Raises

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is 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()**

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_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_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.

**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 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)
   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_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:
XBeDevice.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:
XBeDevice.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.

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()
    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:
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.
ad_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 IMessage

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.
ad_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 IMessage

Raises ValueError – if timeout is less than 0.

reset ()
Override method.
See also:

AbstractXBeeDevice.reset ()

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:

```python
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** *(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:

```python
XBeeDevice.send_bluetooth_data()
XBeeDevice.send_user_data_relay()
```

**send_packet** *(packet, sync=False)*
Override method.

See also:

```python
AbstractXBeeDevice._send_packet()
```

**send_packet\_sync\_and\_get\_response** *(packet\_to\_send)*
Override method.

See also:

```python
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)

Deprecated.

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.

Returns: the serial port associated to the XBee device.

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)

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_dest_address` *(addr)*
Depreciated.

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)*
Depreciated.

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.

• `ATCommandException` – if the response is not as expected.

See also:

`IOLine`
`IOMode`

**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 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.

**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.WiFiDevice` *(port, baud_rate)*

**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.

:raises All exceptions raised by `XBeeDevice.__init__()` constructor:

**See also:**

`XBeeDevice`

`XBeeDevice.__init__()`
open()
Override.

Raises XBeeException – if the protocol is invalid.
:raises All exceptions raised by XBeeDevice.open():

See also:

XBeeDevice.open()

get_protocol()
Override.

See also:

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

customize Nguy

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()
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:
WiFiDevice.connect_by_ap()
WiFiDevice.disconnect()
WiFiDevice.get_access_point()
WiFiDevice.get_access_point_timeout()
WiFiDevice.scan_access_points()
WiFiDevice.set_access_point_timeout()
• **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:

```python
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:

```python
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:

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:

WiFiDevice.set_mask_address()
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:**

```
WiFiDevice.get_mask_address()
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:**

```
WiFiDevice.set_dns_address()
ipaddress.IPv4Address
```

**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()
ipaddress.IPv4Address
```

**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()
```
```
ipaddress.IPv4Address
```

**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()
```
```
ipaddress.IPv4Address
```

**add_bluetooth_data_received_callback**(callback)

Override.

**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)

Override.

**add_ip_data_received_callback**(callback)

Adds a callback for the event IPDataReceived.

**Parameters**
callback (Function) – the callback. Receives one argument.

• The data received as an IPMessage

**add_micropython_data_received_callback**(callback)

Override.

**add_modem_status_received_callback**(callback)

Override.

**add_packet_received_callback**(callback)

Override.

**add_user_data_relay_received_callback**(callback)

Override.

**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.

close()
Closes the communication with the XBee device.
This method guarantees that all threads running are stopped and the serial port is closed.

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)
Override.

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)
Override.

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_micropython_data_received_callback(callback)
   Override.

del_modem_status_received_callback(callback)
   Override.

del_packet_received_callback(callback)
   Override.

del_user_data_relay_received_callback(callback)
   Override.

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.

disable_bluetooth()

execute_command(parameter)
   Executes the provided 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.

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 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()

Returns the API output mode of the XBe device.

The API output mode determines the format that the received data is output through the serial interface of the XBe device.

Returns the API output mode of the XBe device.

Return type APIOutputMode

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:

`APIOutputMode`

**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 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

google_firmware_version()
Returns the firmware version of the XBee device.

Returns the hardware version of the XBee device.

Return type bytearray

google_hardware_version()
Returns the hardware version of the XBee device.

Returns the hardware version of the XBee device.

Return type HardwareVersion

See also:

HardwareVersion

google_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)
Override.
See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee device.

Parameters

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_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 \texttt{OperatingMode}. This XBee device’s operating mode.

\textbf{read\_data} (timeout=None, explicit=False)

Deprecated.

Operation not supported in this protocol. This method will raise an \texttt{AttributeError}.

\textbf{read\_data\_from} (remote_xbee_device, timeout=None, explicit=False)

Deprecated.

Operation not supported in this protocol. This method will raise an \texttt{AttributeError}.

\textbf{read\_device\_info}()

Override.

See also:

\texttt{AbstractXBeeDevice.read_device\_info()}

\textbf{read\_io\_sample}()

Returns an IO sample from the XBee device containing the value of all enabled digital IO and analog input channels.

\textbf{Returns} the IO sample read from the XBee device.

\textbf{Return type} \texttt{IOSample}

\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.

See also:

\texttt{IOSample}

\textbf{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 \texttt{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 \texttt{IPDevice.start\_listening()} for that purpose. When finished, you can use the method \texttt{IPDevice.stop\_listening()} to stop listening for incoming IP data.

\textbf{Parameters} \texttt{timeout} (Integer, optional) – The time to wait for new IP data in seconds.

\textbf{Returns} IP message, \texttt{None} if this device did not receive new data.

\textbf{Return type} \texttt{IPMessage}
Raises `ValueError` if `timeout` is less than 0.

```python
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.

```python
reset()
```
Override method.

See also:

```
AbstractXBeeDevice.reset()
```

```python
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`.
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.
• **OperationNotSupportedError** – 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`))

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

- \textit{local\_interface} (XBeeLocalInterface) – Destination XBee local interface.
- \textit{data} (Bytearray) – Data to send.

Raises

- \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{ValueError} – if \textit{local\_interface} is None.
- \texttt{XBeeException} – if there is any problem sending the User Data Relay.

See also:

\texttt{XBeeLocalInterface}

serial_port

Returns the serial port associated to the XBee device.

\begin{itemize}
  \item Returns the serial port associated to the XBee device.
  \item \texttt{XBeeSerialPort}
\end{itemize}

See also:

\texttt{XBeeSerialPort}

set_16bit_addr \((value)\)

Sets the 16-bit address of the XBee device.

Parameters \texttt{value} (XBee16BitAddress) – the new 16-bit address of the XBee device.

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 current protocol is not 802.15.4.

set_api_output_mode \((api\_output\_mode)\)

Sets the API output mode of the XBee device.

Parameters \texttt{api\_output\_mode} (APIOutputMode) – the new API output mode of the XBee device.

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 current protocol is ZigBee

See also:

\[APIOutputMode\]

**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:
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.

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.RemoteXBeeDevice:
    # ...
```

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

```python
def get_parameter(parameter):
    # ...
```

Override. See also:

- `AbstractXBeeDevice.get_parameter()`

```python
def 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

**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.
**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.

*Raises*

- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBee device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBee device’s operating mode is 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()

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_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.
• \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.

\texttt{\textbf{get_node_id}()}

Returns the Node Identifier (\texttt{NI}) value of the XBee device.

\textbf{Returns} the Node Identifier (\texttt{NI}) of the XBee device.

\textbf{Return type} \texttt{String}

\texttt{\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} \texttt{Bytearray}

\textbf{Raises} \textit{TimeoutException} – if the response is not received before the read timeout expires.

\texttt{\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} \textit{TimeoutException} – if the response is not received before the read timeout expires.

\textbf{See also:}

\texttt{PowerLevel}

\texttt{\textbf{get_protocol}()}

Returns the current protocol of the XBee device.

\textbf{Returns} the current protocol of the XBee device.

\textbf{Return type} \texttt{XBeeProtocol}

\textbf{See also:}

\texttt{XBeeProtocol}

\texttt{\textbf{get_pwm_duty_cycle}(io\_line)}

Returns the PWM duty cycle in \% corresponding to the provided IO line.

\textbf{Parameters} \texttt{io\_line (IOLine)} – the IO line to get its PWM duty cycle.

\textbf{Returns} the PWM duty cycle of the given IO line or \texttt{None} if the response is empty.

\textbf{Return type} \texttt{Integer}

\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.
• `InvalidOperatingModeException` – if the XBee device’s operating mode is 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_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

`read_device_info()`
Updates all instance parameters reading them from 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.

`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

\texttt{set\_16bit\_addr}(value)

Sets the 16-bit address of the XBee device.

\textbf{Parameters} \texttt{value} (XBee16BitAddress) -- the new 16-bit address of the XBee device.

\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{OperationNotSupportedException} -- if the current protocol is not 802.15.4.

\texttt{set\_api\_output\_mode}(api\_output\_mode)

Sets the API output mode of the XBee device.

\textbf{Parameters} \texttt{api\_output\_mode} (APIOutputMode) -- the new API output mode of the XBee device.

\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{OperationNotSupportedException} -- if the current protocol is ZigBee

See also:

APIOutputMode

\texttt{set\_dest\_address}(addr)

Sets the 64-bit address of the XBee device that data will be reported to.

\textbf{Parameters} \texttt{addr} (XBee64BitAddress or RemoteXBeeDevice) -- the address itself or the remote XBee device that you want to set up its address as destination address.

\textbf{Raises} \texttt{TimeoutException} -- if the response is not received before the read timeout expires.

:raises All exceptions raised by XBeeDevice.set_parameter():

\texttt{set\_dio\_change\_detection}(io\_lines\_set)

Sets the digital IO lines to be monitored and sampled whenever their status changes.

A \texttt{None} set of lines disables this feature.

\textbf{Parameters} \texttt{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:
The XBee Python Library Documentation, Release 1.2.0

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*)
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.

**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.

:raises All exceptions raised by RemoteXBeeDevice constructor.: 

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 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.

disable_bluetooth()
Disables the Bluetooth interface of this XBe 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 XBe 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.

eexecute_command(parameter)
Executes the provided 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.

gget_16bit_addr()
Returns the 16-bit address of the XBe device.

Returns the 16-bit address of the XBe device.
Return type  
XBee6BitAddress

See also:

XBee6BitAddress

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()

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_bluetooth_mac_addr()`
Reads and returns the EUI-48 Bluetooth MAC address of this XBees 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 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.

`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 XBees 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)`
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_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

read_device_info()
Updates all instance parameters reading them from 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.

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()

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)*

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_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.

::raises All exceptions raised by `XBeeDevice.set_parameter()`.

**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 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(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.

**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.RemoteDigiMeshDevice(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.

    :raises All exceptions raised by `RemoteXBeeDevice` constructor:

    See also:

    RemoteXBeeDevice
    XBee64BitAddress
    XBeeDevice

    `get_protocol()`
    Override.

    See also:

    RemoteXBeeDevice.get_protocol()

    `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.

**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.

**Raises**

• *TimeoutException* – if the response is not received before the read timeout expires.

• *XBeeException* – if the XBee device’s serial port is closed.

• *InvalidOperatingModeException* – if the XBee device’s operating mode is 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:

*XBeel16BitAddress*
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 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()

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 XBe device.

Returns: the API output mode of the XBe device.

Return type: APIOutputMode

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:

*APIOutputMode*

**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** `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 **
XBee Python Library Documentation, Release 1.2.0

- *TimeoutException* – if the response is not received before the read timeout expires.
- *XBeeException* – if the XBee device’s serial port is closed.
- *InvalidOperatingModeException* – if the XBee device’s operating mode is 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)**

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_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

**read_device_info()**
Updates all instance parameters reading them from 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.

`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()`

`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)`

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_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.

:raises All exceptions raised by `XBeeDevice.set_parameter()`.

`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.
2.5. API reference

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_local_xbee_device(local_xbee_device)
This methods associates a XBeeDevice to the remote XBees 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:

```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.

### `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.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.

:raises All exceptions raised by RemoteXBeeDevice constructor.:

See also:

RemoteXBeeDevice
XBee64BitAddress
XBeeDevice

get_protocol()
Override.

See also:

RemoteXBeeDevice.get_protocol()

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.

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.
- `ATCommandException` – if the response is not as expected.

`execute_command(parameter)`
Executes the provided 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.

`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 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()

Returns the API output mode of the XBe device.

The API output mode determines the format that the received data is output through the serial interface of the XBe device.

Returns

the API output mode of the XBe device.

Return type

APIOutputMode

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:

APIOutputMode
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  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.
**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)**

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_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

**read_device_info()**

Updates all instance parameters reading them from 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.

`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()`

`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)`

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 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_dest_address (addr)
Sets the 64-bit address of the XBees device that data will be reported to.

Parameters addr (XBee64BitAddress or RemoteXBeeDevice) – the address itself or the remote XBees 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.

:raises All exceptions raised by XBeeDevice.set_parameter() :

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 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

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 XBee device’s operating mode is 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`

```python
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`

```python
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.

```python
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:

2.5. API reference
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.

**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.

### 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.

:raises All exceptions raised by RemoteXBeeDevice constructor:

See also:

RemoteXBeeDevice
XBee16BitAddress
XBee64BitAddress
XBeeDevice

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()
apply_changes()
Applies changes via AC command.

Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBeex device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeex device’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – if the response is not as expected.

disable_bluetooth()
Disables the Bluetooth interface of this XBeex 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 XBeex device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeex 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 XBeex 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 XBeex device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeex 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.

Raises
- `TimeoutException` – if the response is not received before the read timeout expires.
- `XBeeException` – if the XBeex device’s serial port is closed.
- `InvalidOperatingModeException` – if the XBeex device’s operating mode is 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 XBeex device.

Returns the 16-bit address of the XBeex device.
Return type: XBee6BitAddress

See also:

XBee6BitAddress

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_iow_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()

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 XBeet device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBeet device’s operating mode is 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_bluetooth_mac_addr()**

Reads and returns the EUI-48 Bluetooth MAC address of this XBeet 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 XBeet device’s serial port is closed.
- **InvalidOperatingModeException** – if the XBeet 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 XBeet 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.
• \textit{OperationNotSupportedException} – if the received data is not an IO mode.

\textbf{get\_io\_sampling\_rate()} \\
Returns the IO sampling rate of the XBee device.

\begin{itemize}
\item \textbf{Returns} the IO sampling rate of XBee device.
\item \textbf{Return type} Integer
\item \textbf{Raises}
\begin{itemize}
\item \textit{TimeoutException} – if the response is not received before the read timeout expires.
\item \textit{XBeeException} – if the XBee device’s serial port is closed.
\item \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.
\item \textit{ATCommandException} – if the response is not as expected.
\end{itemize}
\end{itemize}

\textbf{get\_local\_xbee\_device()} \\
Returns the local XBee device associated to the remote one.

\begin{itemize}
\item \textbf{Returns} \textit{XBeeDevice}
\end{itemize}

\textbf{get\_node\_id()} \\
Returns the Node Identifier (NI) value of the XBee device.

\begin{itemize}
\item \textbf{Returns} the Node Identifier (NI) of the XBee device.
\item \textbf{Return type} String
\end{itemize}

\textbf{get\_pan\_id()} \\
Returns the operating PAN ID of the XBee device.

\begin{itemize}
\item \textbf{Returns} operating PAN ID of the XBee device.
\item \textbf{Return type} Bytearray
\item \textbf{Raises} \textit{TimeoutException} – if the response is not received before the read timeout expires.
\end{itemize}

\textbf{get\_parameter\( (\text{parameter})\)()} \\
Override.

\textbf{See also:}

\begin{itemize}
\item AbstractXBeeDevice.get\_parameter() \end{itemize}

\textbf{get\_power\_level()} \\
Returns the power level of the XBee device.

\begin{itemize}
\item \textbf{Returns} the power level of the XBee device.
\item \textbf{Return type} \textit{PowerLevel}
\item \textbf{Raises} \textit{TimeoutException} – if the response is not received before the read timeout expires.
\item \textbf{See also:}
\end{itemize}

\begin{itemize}
\item PowerLevel \end{itemize}
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_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

**read_device_info()**

Updates all instance parameters reading them from 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.

**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()`

**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.

`sset_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

`sset_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.

:raises All exceptions raised by `XBeeDevice.set_parameter()`.

`sset_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 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**(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.

**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.

`start_discovery_process()`
Starts the discovery process. This method is not blocking.

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.

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_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()

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()

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_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.

`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 0x20 and 0xFF
get_device_by_64 (x64bit_addr)
Returns the remote device already contained 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 remote XBee device in the network or None if it is not found.

Return type RemoteXBeeDevice

Raises ValueError – if x64bit_addr is None or unknown.

get_device_by_16 (x16bit_addr)
Returns the remote device already contained 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 remote XBee device in the network or None if it is not found.

Return type RemoteXBeeDevice

Raises ValueError – if x16bit_addr is None or unknown.

get_device_by_node_id (node_id)
Returns the remote device already contained 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 remote XBee device in the network or None if it is not found.

Return type RemoteXBeeDevice

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 RemoteXBeeDevice

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 param-
eters 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 de-
vice ops.

Return type  Tuple (Function, Function)

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_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** RemoteXBeeDevice

**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()
```

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_device_by_16 *(x16bit_addr)*
Returns the remote device already contained 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 remote XBee device in the network or **None** if it is not found.
Return type `RemoteXBeeDevice`

Raises `ValueError` – if `x16bit_addr` is `None` or unknown.

def get_device_by_64(x64bit_addr)
    Returns the remote device already contained 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 remote XBee device in the network or `None` if it is not found.

    Return type `RemoteXBeeDevice`

    Raises `ValueError` – if `x64bit_addr` is `None` or unknown.

def get_device_by_node_id(node_id)
    Returns the remote device already contained 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 remote XBee device in the network or `None` if it is not found.

    Return type `RemoteXBeeDevice`

    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 de-
    vice 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_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.

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 0x20 and 0xFF

start_discovery_process ()
Starts the discovery process. This method is not blocking.

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.

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.

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_discovery_process_finished_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 `RemoteXBeeDevice`

`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()*

**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_device_by_16** (*x16bit_addr*)

Returns the remote device already contained 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 remote XBee device in the network or **None** if it is not found.

Return type **RemoteXBeeDevice**

Raises **ValueError** – if **x16bit_addr** is **None** or unknown.

**get_device_by_64** (*x64bit_addr*)

Returns the remote device already contained 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 remote XBee device in the network or **None** if it is not found.

Return type **RemoteXBeeDevice**

Raises **ValueError** – if **x64bit_addr** is **None** or unknown.

**get_device_by_node_id** (*node_id*)

Returns the remote device already contained 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 remote XBee device in the network or **None** if it is not found.

Return type **RemoteXBeeDevice**

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_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.

`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 0x20 and 0xFF

start_discovery_process ()
Starts the discovery process. This method is not blocking.

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.

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.
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

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_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.
### add_remote

**Return type** `RemoteXBeeDevice`

`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

`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

`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

`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()`
**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_device_by_16** (*x16bit_addr*)

Returns the remote device already contained 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 remote XBee device in the network or *None* if it is not found.

**Return type** *RemoteXBeeDevice*

**Raises** *ValueError* – if *x16bit_addr* is *None* or unknown.

**get_device_by_64** (*x64bit_addr*)

Returns the remote device already contained 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 remote XBee device in the network or *None* if it is not found.

**Return type** *RemoteXBeeDevice*

**Raises** *ValueError* – if *x64bit_addr* is *None* or unknown.

**get_device_by_node_id** (*node_id*)

Returns the remote device already contained 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 remote XBee device in the network or *None* if it is not found.

**Return type** *RemoteXBeeDevice*

**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 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.

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 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.

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.

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 0x20 and 0xFF

start_discovery_process ()
Starts the discovery process. This method is not blocking.

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.

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.

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_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 RemoteXBeeDevice

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()`

### 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_device_by_16(`x16bit_addr`)

Returns the remote device already contained 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 remote XBee device in the network or `None` if it is not found.
- Return type `RemoteXBeeDevice`
- Raises `ValueError` – if `x16bit_addr` is `None` or unknown.

### get_device_by_64(`x64bit_addr`)

Returns the remote device already contained 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 remote XBee device in the network or `None` if it is not found.
- Return type `RemoteXBeeDevice`
- Raises `ValueError` – if `x64bit_addr` is `None` or unknown.

### get_device_by_node_id(`node_id`)

Returns the remote device already contained 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 remote XBee device in the network or **None** if it is not found.

Return type **RemoteXBeeDevice**

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_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.

**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 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 `discovery_timeout` is not between 0x20 and 0xFF

**start_discovery_process()**

Starts the discovery process. This method is not blocking.

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.

**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.

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`

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='The operating mode of the XBee device is not supported by the library.'

    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.

    classmethod from_operating_mode(operating_mode)
        Class constructor.

        Parameters operating_mode (OperatingMode) – the operating mode that generates the exceptions.

    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)')

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.

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(code)
Bases: enum.Enum
Enumerates the possible values of a IOLine configured as digital I/O.
code
Integer. The IO value code.
class digi.xbee.io.IOSample(io_sample_payload)
Bases: object
This class represents an IO Data Sample. The sample is built using the 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:

<table>
<thead>
<tr>
<th>bit 0</th>
<th>bit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO01</td>
<td>DIO10</td>
</tr>
</tbody>
</table>

(continues on next page)
Digital Channel Mask Indicates which digital IO lines are configured in the module. Each bit corresponds to one digital IO line on the module:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DIO0AD0</td>
</tr>
<tr>
<td>1</td>
<td>DIO1AD1</td>
</tr>
<tr>
<td>2</td>
<td>DIO2AD2</td>
</tr>
<tr>
<td>3</td>
<td>DIO3AD3</td>
</tr>
<tr>
<td>4</td>
<td>DIO4AD4</td>
</tr>
<tr>
<td>5</td>
<td>DIO5AD5ASSOC</td>
</tr>
<tr>
<td>6</td>
<td>DIO6RTS</td>
</tr>
<tr>
<td>7</td>
<td>DIO7CTS</td>
</tr>
<tr>
<td>8</td>
<td>DIO8DTRSLPPE_RQ</td>
</tr>
<tr>
<td>9</td>
<td>DIO9ON_SLEEPS</td>
</tr>
<tr>
<td>10</td>
<td>DIO10PWM0RSSI</td>
</tr>
<tr>
<td>11</td>
<td>DIO11PWM1</td>
</tr>
<tr>
<td>12</td>
<td>DIO12CD</td>
</tr>
<tr>
<td>13</td>
<td>DIO13</td>
</tr>
<tr>
<td>14</td>
<td>DIO14</td>
</tr>
<tr>
<td>15</td>
<td>NA</td>
</tr>
</tbody>
</table>

Example: mask of 0x040B means DIO0, DIO1, DIO2, DIO3 and DIO10 enabled.

0 0 0 0 0 0 1 0 0 1 0 0 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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AD0DIO0</td>
</tr>
<tr>
<td>1</td>
<td>AD1DIO1</td>
</tr>
<tr>
<td>2</td>
<td>AD2DIO2</td>
</tr>
<tr>
<td>3</td>
<td>AD3DIO3</td>
</tr>
<tr>
<td>4</td>
<td>AD4DIO4</td>
</tr>
<tr>
<td>5</td>
<td>AD5DIO5ASSOC</td>
</tr>
<tr>
<td>6</td>
<td>NA</td>
</tr>
<tr>
<td>7</td>
<td>Supply Voltage Value</td>
</tr>
</tbody>
</table>

Example: mask of 0x83 means AD0, and AD1 enabled.

0 0 0 0 0 0 0 1

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.

*Return type* `IOValue`

See also:

`IOLine`

`IOValue`
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

digi.xbee.reader module

class digi.xbee.reader.XBeeEvent
    Bases: list
    This class represents a generic XBee event.
    New event callbacks can be added here following this prototype:

    >>> def callback_prototype(*args, **kwargs):
    >>>     #do something...

    All of them will be executed when the event is fired.
    
    See also:

    list (Python standard class)

    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.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 XBees 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:
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.

    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.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.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.

```python
class digi.xbee.reader.ExplicitDataReceived
    Bases: digi.xbee.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** (*SMSMessage*): message containing the phone number that sent the message and the content (data) of the message.

See also:

*XBeeEvent*

*SMSMessage*

**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

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

**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.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.

raise ValueError if the value is not present.

insert()  
Insert object before index.

pop()  
Remove and return item at index (default last).

raise IndexError if list is empty or index is out of range.

remove()  
Remove first occurrence of value.

raise ValueError if the value is not present.

reverse()  
Reverse IN PLACE.

sort()  
Stable sort IN PLACE.

class digi.xbee.reader.PacketListener(serial_port, 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. 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.

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. 1.2 sender (RemoteXBeedevice): the remote XBee device who has sent the 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.
• **serial_port** (XbeeSerialPort) – the COM port to which this listener will be listening.

• **xbee_device** (XBeeDevice) – the XBee that is the listener owner.

• **queue_max_size** (Integer) – the maximum size of the XBee queue.

**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) – the callback. Receives two arguments.

  • The received packet as a XBeeAPIPacket

  • The sender as a RemoteXBeeDevice

**add_data_received_callback** (**callback**)  
Adds a callback for the event DataReceived.

  Parameters **callback** (Function) – the callback. Receives one argument.

  • The data received as an XBeeMessage

**add_modem_status_received_callback** (**callback**)  
Adds a callback for the event ModemStatusReceived.

2.5. API reference
Parameters `callback (Function)` – the callback. Receives one argument.
- The modem status as a `ModemStatus`

`add_io_sample_received_callback (callback)`
Add a callback for the event `IOSampleReceived`.

Parameters `callback (Function)` – 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)`
Add a callback for the event `ExplicitDataReceived`.

Parameters `callback (Function)` – the callback. Receives one argument.
- The explicit data received as an `ExplicitXBeeMessage`

`add_ip_data_received_callback (callback)`
Add a callback for the event `IPDataReceived`.

Parameters `callback (Function)` – the callback. Receives one argument.
- The data received as an `IPMessage`

`add_sms_received_callback (callback)`
Add a callback for the event `SMSReceived`.

Parameters `callback (Function)` – the callback. Receives one argument.
- The data received as an `SMSMessage`

`add_user_data_relay_received_callback (callback)`
Add a callback for the event `RelayDataReceived`.

Parameters `callback (Function)` – the callback. Receives one argument.
- The data received as a `UserDataRelayMessage`

`add_bluetooth_data_received_callback (callback)`
Add a callback for the event `BluetoothDataReceived`.

Parameters `callback (Function)` – the callback. Receives one argument.
- The data received as a Bytearray

`add_micropython_data_received_callback (callback)`
Add a callback for the event `MicroPythonDataReceived`.

Parameters `callback (Function)` – the callback. Receives one argument.
- The data received as a Bytearray

`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.

**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.

**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 \((\text{block=True, timeout=None})\)

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 \((\text{remote}_x\text{bee}_\text{device}, \text{timeout=None})\)

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 first 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 \((\text{ip_addr, timeout=None})\)

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.

```python
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

- `None` 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

```python
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.

```python
flush()
```

Clears the queue.

```python
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.

```python
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.

```python
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.serial module

class digi.xbee.serial.FlowControl

    Bases: enum.Enum

    This class represents all available flow controls.

class digi.xbee.serial.XBeeSerialPort *(baud_rate, port, 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>, timeout=0.1)*

    Bases: sphinx.ext.autodoc.importer._MockObject

    This class extends the functionality of Serial class (PySerial).

    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

**read_byte** ()

Synchronous. Reads one byte from serial port.

- **Returns** the read byte.

- **Return type** `Integer`

- **Raises** `TimeoutException` – if there is no bytes in 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`.

**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.
CHAPTER 3

Indices and tables

- genindex
- modindex
- search
Copyright 2017, 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, 94
digi.xbee, 94
digi.xbee.devices, 235
digi.xbee.exception, 526
digi.xbee.io, 528
digi.xbee.models, 94
digi.xbee.models.accesspoint, 94
digi.xbee.models.address, 101
digi.xbee.models.atcomm, 96
digi.xbee.models.hw, 97
digi.xbee.models.message, 103
digi.xbee.models.mode, 99
digi.xbee.models.options, 107
digi.xbee.models.protocol, 112
digi.xbee.models.status, 113
digi.xbee.packets, 121
digi.xbee.packets.aft, 121
digi.xbee.packets.base, 122
digi.xbee.packets.cellular, 131
digi.xbee.packets.common, 136
digi.xbee.packets.devicecloud, 174
digi.xbee.packets.factory, 231
digi.xbee.packets.network, 190
digi.xbee.packets.raw, 196
digi.xbee.packetsrelay, 216
digi.xbee.packets.wifi, 222
digi.xbee.reader, 533
digi.xbee.serial, 551
digi.xbee.util, 231
digi.xbee.util.utils, 231
Index

A

AbstractXBeeDevice (class in digi.xbee.devices), 235

AccessPoint (class in digi.xbee.models.accesspoint), 94

ADC (digi.xbee.io.IOMode attribute), 532

add_bluetooth_data_received_callback()
(digi.xbee.devices.CellularDevice method), 364

add_bluetooth_data_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 286

add_bluetooth_data_received_callback()
(digi.xbee.devices.DigiPointDevice method), 305

add_bluetooth_data_received_callback()
(digi.xbee.devices.IPDevice method), 347

add_bluetooth_data_received_callback()
(digi.xbee.devices.LPWANDevice method), 381

add_bluetooth_data_received_callback()
(digi.xbee.devices.NBLoTDevice method), 398

add_bluetooth_data_received_callback()
(digi.xbee.devices.Raw802Device method), 267

add_bluetooth_data_received_callback()
(digi.xbee.devices.WiFiDevice method), 422

add_bluetooth_data_received_callback()
(digi.xbee.devices.XBeeDevice method), 253

add_data_received_callback()
(digi.xbee.devices.CellularDevice method), 364

add_data_received_callback()
(digi.xbee.devices.DigiMeshDevice method), 286

add_data_received_callback()
(digi.xbee.devices.DigiPointDevice method), 305

add_data_received_callback()
(digi.xbee.devices.IPDevice method), 347

add_data_received_callback()
(digi.xbee.devices.LPWANDevice method), 381

add_data_received_callback()
(digi.xbee.devices.NBLoTDevice method), 398

add_data_received_callback()
(digi.xbee.devices.Raw802Device method), 267

add_data_received_callback()
(digi.xbee.devices.WiFiDevice method), 422

add_data_received_callback()
(digi.xbee.devices.XBeeNetwork method), 501

add_device_discovered_callback()
(digi.xbee.devices.DigiMeshNetwork method), 516

add_device_discovered_callback()
(digi.xbee.devices.DigiPointNetwork method), 521

add_device_discovered_callback()
(digi.xbee.devices.Raw802Network method), 510

add_device_discovered_callback()
(digi.xbee.devices.XBeeNetwork method), 501

add_device_discovered_callback()
(digi.xbee.devices.ZigBeeNetwork method), 505

add_discovery_process_finished_callback()
(digi.xbee.devices.DigiMeshNetwork method), 516

559
add_discovery_process_finished_callback() 
(digi.xbee.devices.DigiPointNetwork method), 521
add_discovery_process_finished_callback() 
(digi.xbee.devices.Raw802Network method), 511
add_discovery_process_finished_callback() 
(digi.xbee.devices.XBeeNetwork method), 501
add_discovery_process_finished_callback() 
(digi.xbee.devices.ZigBeeNetwork method), 505
add_explicit_data_received_callback() 
(digi.xbee.devices.CellularDevice method), 364
add_explicit_data_received_callback() 
(digi.xbee.devices.DigiMeshDevice method), 286
add_explicit_data_received_callback() 
(digi.xbee.devices.DigiPointDevice method), 305
add_explicit_data_received_callback() 
(digi.xbee.devices.IPDevice method), 347
add_explicit_data_received_callback() 
(digi.xbee.devices.LPWANDevice method), 381
add_explicit_data_received_callback() 
(digi.xbee.devices.NBIoTDevice method), 398
add_explicit_data_received_callback() 
(digi.xbee.devices.Raw802Device method), 267
add_explicit_data_received_callback() 
(digi.xbee.devices.WiFiDevice method), 422
add_ip_data_received_callback() 
(digi.xbee.devices.CellularDevice method), 364
add_ip_data_received_callback() 
(digi.xbee.devices.IPDevice method), 344
add_ip_data_received_callback() 
(digi.xbee.devices.LPWANDevice method), 381
add_ip_data_received_callback() 
(digi.xbee.devices.NBIoTDevice method), 398
add_ip_data_received_callback() 
(digi.xbee.devices.WiFiDevice method), 422
add_micropython_data_received_callback() 
(digi.xbee.devices.CellularDevice method), 364
add_micropython_data_received_callback() 
(digi.xbee.devices.DigiMeshDevice method), 286
add_micropython_data_received_callback() (digi.xbee.devices.DigiPointDevice method), 305
add_micropython_data_received_callback() (digi.xbee.devices.IPDevice method), 347
add_micropython_data_received_callback() (digi.xbee.devices.LPWANDevice method), 381
add_micropython_data_received_callback() (digi.xbee.devices.NBIoTDevice method), 399
add_micropython_data_received_callback() (digi.xbee.devices.Raw802Device method), 267
add_micropython_data_received_callback() (digi.xbee.devices.XBeeDevice method), 253
add_micropython_data_received_callback() (digi.xbee.devices.ZigBeeDevice method), 326
add_modem_status_received_callback() (digi.xbee.devices.CellularDevice method), 364
add_modem_status_received_callback() (digi.xbee.devices.DigiMeshDevice method), 286
add_modem_status_received_callback() (digi.xbee.devices.DigiPointDevice method), 305
add_modem_status_received_callback() (digi.xbee.devices.IPDevice method), 347
add_modem_status_received_callback() (digi.xbee.devices.LPWANDevice method), 381
add_modem_status_received_callback() (digi.xbee.devices.NBIoTDevice method), 399
add_modem_status_received_callback() (digi.xbee.devices.Raw802Device method), 267
add_modem_status_received_callback() (digi.xbee.devices.XBeeDevice method), 253
add_modem_status_received_callback() (digi.xbee.devices.ZigBeeDevice method), 326
add_modem_status_received_callback() (digi.xbee.reader.PacketListener method), 546
add_modem_status_received_callback() (digi.xbee.devices.CellularDevice method), 364
add_modem_status_received_callback() (digi.xbee.devices.DigiMeshDevice method), 286
add_modem_status_received_callback() (digi.xbee.devices.DigiPointDevice method), 305
add_modem_status_received_callback() (digi.xbee.devices.IPDevice method), 347
add_modem_status_received_callback() (digi.xbee.devices.LPWANDevice method), 381
add_modem_status_received_callback() (digi.xbee.devices.NBIoTDevice method), 399
add_modem_status_received_callback() (digi.xbee.devices.Raw802Device method), 267
add_modem_status_received_callback() (digi.xbee.devices.XBeeDevice method), 253
add_modem_status_received_callback() (digi.xbee.devices.ZigBeeDevice method), 326
add_sms_callback() (digi.xbee.devices.CellularDevice method), 362
add_sms_callback() (digi.xbee.devices.LPWANDevice method), 381
add_sms_callback() (digi.xbee.devices.NBIoTDevice method), 399
add_sms_received_callback() (digi.xbee.devices.DigiMeshDevice method), 286
add_user_data_relay_received_callback() (digi.xbee.devices.CellularDevice method), 364
add_user_data_relay_received_callback() (digi.xbee.devices.LPWANDevice method), 381
add_user_data_relay_received_callback() (digi.xbee.devices.NBIoTDevice method), 399
add_user_data_relay_received_callback() (digi.xbee.devices.DigiPointDevice method), 305
add_user_data_relay_received_callback() (digi.xbee.devices.IPDevice method), 347
add_user_data_relay_received_callback() (digi.xbee.devices.Raw802Device method), 267
add_user_data_relay_received_callback() (digi.xbee.devices.XBeeDevice method), 253
add_user_data_relay_received_callback() (digi.xbee.devices.ZigBeeDevice method), 326
add_remotes() (digi.xbee.devices.DigiMeshNetwork method), 517
add_remotes() (digi.xbee.devices.DigiPointNetwork method), 522
add_remotes() (digi.xbee.devices.Raw802Network method), 511
add_remotes() (digi.xbee.devices.XBeeNetwork method), 504
add_remotes() (digi.xbee.devices.ZigBeeNetwork method), 506
add_remotes() (digi.xbee.devices.DigiMeshNetwork method), 517
add_remotes() (digi.xbee.devices.DigiPointNetwork method), 522
add_remotes() (digi.xbee.devices.Raw802Network method), 511
add_remotes() (digi.xbee.devices.XBeeNetwork method), 505
add_remotes() (digi.xbee.devices.ZigBeeNetwork method), 506
add_sms_callback() (digi.xbee.devices.LPWANDevice method), 381
add_sms_callback() (digi.xbee.devices.NBIoTDevice method), 399
add_sms_received_callback() (digi.xbee.reader.PacketListener method), 546
add_sms_received_callback() (digi.xbee.reader.PacketListener method), 546
add_sms_received_callback() (digi.xbee.reader.PacketListener method), 546
add_user_data_relay_received_callback() (digi.xbee.devices.CellularDevice method), 364
add_user_data_relay_received_callback()
    (digi.xbee.devices.DigiMeshDevice method), 286
add_user_data_relay_received_callback()
    (digi.xbee.devices.DigiPointDevice method), 305
add_user_data_relay_received_callback()
    (digi.xbee.devices.IPDevice method), 347
add_user_data_relay_received_callback()
    (digi.xbee.devices.LPWANDevice method), 381
add_user_data_relay_received_callback()
    (digi.xbee.devices.NBIoTDevice method), 399
add_user_data_relay_received_callback()
    (digi.xbee.devices.Raw802Device method), 267
add_user_data_relay_received_callback()
    (digi.xbee.devices.WiFiDevice method), 422
add_user_data_relay_received_callback()
    (digi.xbee.devices.XBeeDevice method), 253
add_user_data_relay_received_callback()
    (digi.xbee.devices.ZigBeeDevice method), 326
add_user_data_relay_received_callback(
    (digi.xbee.devices.ZigBeeDevice method), 326
address (digi.xbee.models.address.XBee16BitAddress attribute), 102
address (digi.xbee.models.address.XBee64BitAddress attribute), 103
address (digi.xbee.models.address.XBeeIMEIAddress attribute), 103
analog_mask (digi.xbee.io.IOSample attribute), 532
analog_values (digi.xbee.io.IOSample attribute), 532

ApiFrameType (class in digi.xbee.packets.common), 121
APILoutputMode (class in digi.xbee.models.mode), 100
append() (digi.xbee.reader.BluetoothDataReceived method), 543
append() (digi.xbee.reader.DataReceived method), 535
append() (digi.xbee.reader.XBeeDevice method), 537
append() (digi.xbee.reader.DiscoveryProcessFinished method), 538
append() (digi.xbee.reader.ExplicitDataReceived method), 539
append() (digi.xbee.reader.IOSampleReceived method), 536
append() (digi.xbee.reader.IPDataReceived method), 540
append() (digi.xbee.reader.MicroPythonDataReceived method), 543
append() (digi.xbee.reader.ModemStatusReceived method), 535
append() (digi.xbee.reader.PacketReceived method), 534
append() (digi.xbee.reader.RelayDataReceived method), 542
append() (digi.xbee.reader.SMSSent method), 541
append() (digi.xbee.reader.XBeeEvent method), 533
APPEND_DD (digi.xbee.models.options.DiscoveryOptions attribute), 110
APPEND_RSSI (digi.xbee.models.options.DiscoveryOptions attribute), 110
APPLY_CHANGES (digi.xbee.models.options.RemoteATCmdOptions attribute), 110
apply_changes() (digi.xbee.devices.AbstractXBeeDevice method), 237
apply_changes() (digi.xbee.devices.CellularDevice method), 364
apply_changes() (digi.xbee.devices.DigiMeshDevice method), 286
apply_changes() (digi.xbee.devices.DigiPointDevice method), 305
apply_changes() (digi.xbee.devices.IPDevice method), 348
apply_changes() (digi.xbee.devices.LPWANDevice method), 381
apply_changes() (digi.xbee.devices.NBIoTDevice method), 399
apply_changes() (digi.xbee.devices.Raw802Device method), 267
apply_changes() (digi.xbee.devices.WiFiDevice method), 422
apply_changes() (digi.xbee.devices.XBeeDevice method), 254
apply_changes() (digi.xbee.devices.ZipBeeDevice method), 326
apply_changes() (digi.xbee.devices.RemoteXBeedevice method), 441
apply_changes() (digi.xbee.devices.RemoteZigBeeDevice method), 488
apply_changes() (digi.xbee.devices.WiFiDevice method), 422
apply_changes() (digi.xbee.devices.XBeeDevice method), 254
apply_changes() (digi.xbee.devices.ZipBeeDevice method), 326
APS_ENCRYPTED (digi.xbee.models.options.ReceiveOptions attribute), 107
ascii_to_int() (in module digi.xbee.util.utils), 233
AssociationIndicationStatus (class in digi.xbee.models.status), 117
at_command (digi.xbee.io.IOIOLine attribute), 529
ATCommand (class in digi.xbee.models.atcommand), 96
ATCommandException, 526
ATCommandResponse (class in digi.xbee.models.status), 113
ATCommPacket (class in digi.xbee.packets.common), 562

Index
XBee Python Library Documentation, Release 1.2.0

136
ATCommQueuePacket (class in digi.xbee.packets.common), 139
ATCommResponsePacket (class in digi.xbee.packets.common), 142
ATStringCommand (class in digi.xbee.models.atcomm), 96
BluetoothDataReceived (class in digi.xbee.reader), 542
BROADCAST_ADDRESS (digi.xbee.models.address.XBee16BitAddress attribute), 101
BROADCAST_ADDRESS (digi.xbee.models.address.XBee64BitAddress attribute), 102
BROADCAST_PACKET (digi.xbee.models.options.ReceiveOptions attribute), 107
broadcast_radius (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 168
broadcast_radius (digi.xbee.packets.common.TransmitPacket attribute), 156
build_frame() (in module digi.xbee.packets.factory), 231
bytes_to_int() (in module digi.xbee.util.utils), 233
CellularAssociationIndicationStatus (class in digi.xbee.models.status), 118
CellularDevice (class in digi.xbee.devices), 361
channel (digi.xbee.models.accesspoint.AccessPoint attribute), 95
clear() (digi.xbee.devices.DigiMeshNetwork method), 517
clear() (digi.xbee.devices.DigiPointNetwork method), 522
clear() (digi.xbee.devices.Raw802Network method), 512
clear() (digi.xbee.devices.XBeeNetwork method), 502
clear() (digi.xbee.devices.ZigBeeNetwork method), 506
clear() (digi.xbee.reader.bluetooth.bluetoothdatareceived method), 543
clear() (digi.xbee.reader.datareceived method), 535
clear() (digi.xbee.reader.devicediscovered method), 537
clear() (digi.xbee.reader.discoveryprocessfinished method), 538
clear() (digi.xbee.reader.explicitdatareceived method), 539
clear() (digi.xbee.reader.iOsamplereceived method), 537
clear() (digi.xbee.reader.IPDataReceived method), 540
clear() (digi.xbee.reader.micropythonreceived method), 543
clear() (digi.xbee.reader.modemstatusreceived method), 535
clear() (digi.xbee.reader.packetreceived method), 534
clear() (digi.xbee.reader.relaydatareceived method), 542
clear() (digi.xbee.reader.SMSReceived method), 541
clear() (digi.xbee.reader.XBeevento method), 533
close() (digi.xbee.devices.CellularDevice method), 365
close() (digi.xbee.devices.DigiMeshDevice method), 286
close() (digi.xbee.devices.DigiPointDevice method), 305
close() (digi.xbee.devices.IPDevice method), 348
close() (digi.xbee.devices.LPWANDevice method), 382
close() (digi.xbee.devices.NBIoTDevice method), 399
close() (digi.xbee.devices.Raw802Device method), 267
close() (digi.xbee.devices.WiFiDevice method), 423
close() (digi.xbee.devices.XBeeDevice method), 248
close() (digi.xbee.devices.ZigBeeDevice method), 326
clear_id (digi.xbee.models.message.ExplicitXBeeMessage attribute), 105
clear_id (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 170
clear_id (digi.xbee.packets.common.ExplicitRXIndicatorPacket attribute), 173
code (digi.xbee.io.IOValue attribute), 529
code (digi.xbee.models.accesspoint.WiFiEncryptionType attribute), 95
code (digi.xbee.models.atcomm.SpecialByte attribute), 96
code (digi.xbee.models.hw.HardwareVersion attribute), 99
code (digi.xbee.models.mode.APIOutputMode attribute), 100
code (digi.xbee.models.mode.IPAddressingMode attribute), 100
code (digi.xbee.models.mode.OperatingMode attribute), 100
code (digi.xbee.models.options.DiscoveryOptions attribute), 111
code (digi.xbee.models.options.SendDataRequestOptions attribute), 110
code (digi.xbee.models.options.XBeeLocalInterface attribute), 111
code (digi.xbee.models.protocol.IPProtocol attribute), 113

Index 563
create_xbee_device()
create_xbee_device()
digi.xbee.devices.ZigBeeDevice class method
digi.xbee.devices.XBeeDevice class method
digi.xbee.devices.WiFiDevice class method
digi.xbee.devices.Raw802Device class
digi.xbee.devices.NBIoTDevice class method
digi.xbee.devices.LPWANDevice class
digi.xbee.devices.CellularDevice class
digi.xbee.devices.DigiPointDevice class
digi.xbee.devices.DigiMeshDevice class
digi.xbee.devices.DigiPointDevice class
create_xbee_device()
digi.xbee.devices.DigiMeshDevice class
create_xbee_device()
digi.xbee.devices.IPDevice class method
create_xbee_device()
digi.xbee.devices.CellularDevice class
create_xbee_device()
digi.xbee.devices.DigiMeshDevice class
create_xbee_device()
digi.xbee.devices.LPWANDevice class
create_xbee_device()
create_xbee_device()
create_xbee_device()
create_xbee_device()
create_xbee_device()
daemon (digi.xbee.reader.PacketListener attribute), 547

data (digi.xbee.models.message.ExplicitXBeeMessage attribute), 105
data (digi.xbee.models.message.IPMessage attribute), 106
data (digi.xbee.models.message.SMSMessage attribute), 106
data (digi.xbee.models.message.UserDataRelayMessage attribute), 107
data (digi.xbee.models.message.XBeeMessage attribute), 104
data (digi.xbee.packets.cellular.RXSMSPacket attribute), 132
data (digi.xbee.packets.cellular.TXSMSPacket attribute), 136
data (digi.xbee.packets.network.RXIPv4Packet attribute), 192
data (digi.xbee.packets.network.TXIPv4Packet attribute), 196
data (digi.xbee.packets.relay.UserDataRelayOutputPacket attribute), 221
data (digi.xbee.packets.relay.UserDataRelayPacket attribute), 217

DataReceived (class in digi.xbee.reader), 534
del_bluetooth_data_received_callback() (digi.xbee.devices.CellularDevice method), 365
del_bluetooth_data_received_callback() (digi.xbee.devices.DigiMeshDevice method), 287
del_bluetooth_data_received_callback() (digi.xbee.devices.DigiPointDevice method), 306
del_bluetooth_data_received_callback() (digi.xbee.devices.IPDVDevice method), 348
del_bluetooth_data_received_callback() (digi.xbee.devices.LPWANDevice method), 382
del_bluetooth_data_received_callback() (digi.xbee.devices.NBIoTDevice method), 400
del_bluetooth_data_received_callback() (digi.xbee.devices.Raw802Device method), 268
del_bluetooth_data_received_callback() (digi.xbee.devices.WiFiDevice method), 423
del_bluetooth_data_received_callback() (digi.xbee.devices.XBeeDevice method), 253
del_bluetooth_data_received_callback() (digi.xbee.devices.ZigBeeDevice method), 326
del_bluetooth_data_received_callback() (digi.xbee.devices.ZigBeeNetwork method), 506
del_discovery_process_finished_callback() (digi.xbee.devices.DigiMeshNetwork method), 517
del_discovery_process_finished_callback() (digi.xbee.devices.DigiPointNetwork method), 522
del_discovery_process_finished_callback() (digi.xbee.devices.Raw802Network method), 512
del_discovery_process_finished_callback() (digi.xbee.devices.XBeeNetwork method), 502
del_discovery_process_finished_callback() (digi.xbee.devices.ZigBeeNetwork method), 506
del_discovery_process_finished_callback() (digi.xbee.devices.ZigBeeNetwork method), 517
del_discovery_process_finished_callback() (digi.xbee.devices.DigiPointNetwork method), 522
del_discovery_process_finished_callback() (digi.xbee.devices.Raw802Network method), 512
del_discovery_process_finished_callback() (digi.xbee.devices.XBeeNetwork method), 502
del_discovery_process_finished_callback() (digi.xbee.devices.ZigBeeNetwork method), 506
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>507</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.XBeeDevice method), 253</td>
</tr>
<tr>
<td>365</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.ZigBeeDevice method), 327</td>
</tr>
<tr>
<td>287</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiMeshDevice method), 365</td>
</tr>
<tr>
<td>306</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiPointDevice method), 347</td>
</tr>
<tr>
<td>347</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.IPDevice method), 344</td>
</tr>
<tr>
<td>382</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.LPWANDevice method), 400</td>
</tr>
<tr>
<td>365</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.NBIoTDevice method), 382</td>
</tr>
<tr>
<td>423</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.WiFiDevice method), 268</td>
</tr>
<tr>
<td>423</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.reader.PacketListener method), 547</td>
</tr>
<tr>
<td>268</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.XBeeDevice method), 253</td>
</tr>
<tr>
<td>327</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.ZigBeeDevice method), 327</td>
</tr>
<tr>
<td>287</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiMeshDevice method), 365</td>
</tr>
<tr>
<td>306</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiPointDevice method), 347</td>
</tr>
<tr>
<td>348</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.IPDevice method), 383</td>
</tr>
<tr>
<td>383</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.LPWANDevice method), 400</td>
</tr>
<tr>
<td>400</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.NBIoTDevice method), 382</td>
</tr>
<tr>
<td>268</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.WiFiDevice method), 268</td>
</tr>
<tr>
<td>268</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.reader.PacketListener method), 547</td>
</tr>
<tr>
<td>253</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.XBeeDevice method), 287</td>
</tr>
<tr>
<td>327</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.ZigBeeDevice method), 327</td>
</tr>
<tr>
<td>287</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiMeshDevice method), 365</td>
</tr>
<tr>
<td>306</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiPointDevice method), 347</td>
</tr>
<tr>
<td>344</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.IPDevice method), 382</td>
</tr>
<tr>
<td>400</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.NBIoTDevice method), 382</td>
</tr>
<tr>
<td>268</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.WiFiDevice method), 268</td>
</tr>
<tr>
<td>268</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.reader.PacketListener method), 547</td>
</tr>
<tr>
<td>253</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.XBeeDevice method), 287</td>
</tr>
<tr>
<td>327</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.ZigBeeDevice method), 327</td>
</tr>
<tr>
<td>287</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiMeshDevice method), 365</td>
</tr>
<tr>
<td>306</td>
<td><code>del_exp1_data_received_callback()</code> (digi.xbee.devices.DigiPointDevice method), 347</td>
</tr>
</tbody>
</table>
(digi.xbee.devices.DigiPointDevice method), 306
del_modem_status_received_callback()
(digi.xbee.devices.IPDevice method), 348
del_modem_status_received_callback()
(digi.xbee.devices.LPWANDevice method), 383
del_modem_status_received_callback()
(digi.xbee.devices.NBIoTDevice method), 400
del_modem_status_received_callback()
(digi.xbee.devices.Raw802Device method), 268
del_modem_status_received_callback()
(digi.xbee.devices.WiFiDevice method), 424
del_modem_status_received_callback()
(digi.xbee.devices.XBeeDevice method), 253
del_modem_status_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 327
del_sms_callback()
(digi.xbee.devices.CellularDevice method), 362
del_sms_callback()
(digi.xbee.devices.LPWANDevice method), 400
del_sms_received_callback()
(digi.xbee.devices.LPWANDevice method), 383
del_sms_received_callback()
(digi.xbee.devices.NBIoTDevice method), 400
del_sms_received_callback()
(digi.xbee.devices.Raw802Device method), 268
del_sms_received_callback()
(digi.xbee.devices.WiFiDevice method), 424
del_sms_received_callback()
(digi.xbee.devices.XBeeDevice method), 253
del_sms_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 327
del_sms_received_callback()
(digi.xbee.reader.PacketListener method), 547
del_sms_received_callback()
(digi.xbee.reader.PacketListener method), 547
del_user_data_relay_received_callback()
(digi.xbee.devices.CellularDevice method), 366
del_user_data_relay_received_callback()
(digi.xbee.devices.LPWANDevice method), 383
del_user_data_relay_received_callback()
(digi.xbee.devices.NBIoTDevice method), 400
del_user_data_relay_received_callback()
(digi.xbee.devices.Raw802Device method), 268
del_user_data_relay_received_callback()
(digi.xbee.devices.WiFiDevice method), 424
del_user_data_relay_received_callback()
(digi.xbee.devices.XBeeDevice method), 253
del_user_data_relay_received_callback()
(digi.xbee.devices.ZigBeeDevice method), 327
del_user_data_relay_received_callback()
(digi.xbee.reader.PacketListener method), 547
description (digi.xbee.io.IOLine attribute), 529
description (digi.xbee.models.accesspoint.WiFiEncryptionType attribute), 95
description (digi.xbee.models.hl.hw.HardwareVersion attribute), 99
description (digi.xbee.models.mode.APIOutputMode attribute), 100
description (digi.xbee.models.mode.IPAddressingMode attribute), 101
description (digi.xbee.models.mode.OperatingMode attribute), 100
description (digi.xbee.models.options.DiscoveryOptions attribute), 111
description (digi.xbee.models.options.SendDataRequestOptions attribute), 110
description (digi.xbee.models.options.XBeeLocalInterface attribute), 111
description (digi.xbee.models.protocol.IPProtocol attribute), 113
description (digi.xbee.models.protocol.XBeeProtocol attribute), 112
description (digi.xbee.models.status.AssociationIndicationStatus attribute), 118
 dicity.xbee.exception

dicity.xbee.devices

dicity.xbee

dicity.DictKeys

(dicity.xbee.models.status.ATCommandStatus

diti.xbee.io (module), 528

dicity.xbee.models (module), 94
dicity.xbee.models.status

(dictio.models.CellularAssociationIndicationStatus

diti.xbee.models.accesspoint (module), 94
dicity.xbee.models.address (module), 101
dicity.xbee.models.atcomm (module), 96
dicity.xbee.models.hw (module), 97
dicity.xbee.models.message (module), 103
dicity.xbee.models.mode (module), 99
dicity.xbee.models.options (module), 107
dicity.xbee.models.protocol (module), 112
dicity.xbee.models.status (module), 113
dicity.xbee.models.type (module), 121
dicity.xbee.packets (module), 121

dicity.xbee.packets.aft (module), 121
dicity.xbee.packets.base (module), 122
dicity.xbee.packets.cellular (module), 131
dicity.xbee.packets.common (module), 136
dicity.xbee.packets.devicecloud (module), 174

dicity.xbee.packets.factory (module), 231
dicity.xbee.packets.network (module), 190
dicity.xbee.packets.raw (module), 196
dicity.xbee.packets.relay (module), 216

dicity.xbee.reader (module), 246

dicity.xbee.serial (module), 280

dicity.xbee.packets.relay

dicity.xbee.packets.raw

dicity.xbee.packets.network

dicity.xbee.packets.factory

dicity.xbee.packets.aft

dicity.xbee.packets

dicity.xbee.models.protocol

dicity.xbee.models.mode

dicity.xbee.models.message

dicity.xbee.models.hw

dicity.xbee.models.atcomm

dicity.xbee.models.accesspoint

dicity.xbee.models

(dicity.xbee.models.status.ModemStatus

diti.xbee.models.type (module), 121

dicity.xbee.packets (module), 121

dicity.xbee.models.status

(dicity.xbee.models.status.NetworkDiscoveryStatus

diti.xbee.models.type (module), 121

dicity.xbee.packets (module), 121

dicity.xbee.models.status

(dicity.xbee.models.status.PowerLevelStatus

diti.xbee.models.type (module), 121

dicity.xbee.packets (module), 121

dicity.xbee.models.status

(dicity.xbee.models.status.TransmitStatus

diti.xbee.models.type (module), 121

dicity.xbee.packets (module), 121

dicity.xbee.models.status

(dicity.xbee.models.status.WiFiAssociationIndicationStatus

diti.xbee.models.type (module), 121

dicity.xbee.packets (module), 121

dicity.xbee.models.status

(dicity.xbee.packets.aft.ApiFrameType

diti.xbee.models.type (module), 121

dicity.xbee.packets (module), 121

dicity.xbee.models.skip_at

dicity.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbee.models.skip_at

dicitic.xbe
enable_apply_changes()
(digi.xbee.devices.IPDevice method), 349
enable_apply_changes()
(digi.xbee.devices.LPWANDevice method), 383
enable_apply_changes()
(digi.xbee.devices.NBIoTDevice method), 400
enable_apply_changes()
(digi.xbee.devices.Raw802Device method), 268
enable_apply_changes()
(digi.xbee.devices.RemoteDigiMeshDevice method), 465
enable_apply_changes()
(digi.xbee.devices.RemoteDigiPointDevice method), 477
enable_apply_changes()
(digi.xbee.devices.RemoteRaw802Device method), 453
enable_apply_changes()
(digi.xbee.devices.RemoteXBeeDevice method), 442
enable_apply_changes()
(digi.xbee.devices.RemoteZigBeeDevice method), 489
enable_apply_changes()
(digi.xbee.devices.WiFiDevice method), 424
enable_apply_changes()
(digi.xbee.devices.XBeeDevice method), 255
enable_apply_changes()
(digi.xbee.devices.ZigBeeDevice method), 327
enable_bluetooth()
(digi.xbee.devices.NBIoTDevice method), 400
enable_bluetooth()
(digi.xbee.devices.Raw802Device method), 268
enable_bluetooth()
(digi.xbee.devices.RemoteDigiMeshDevice method), 465
enable_bluetooth()
(digi.xbee.devices.RemoteDigiPointDevice method), 477
enable_bluetooth()
(digi.xbee.devices.RemoteRaw802Device method), 453
enable_bluetooth()
(digi.xbee.devices.RemoteXBeeDevice method), 442
enable_bluetooth()
(digi.xbee.devices.RemoteZigBeeDevice method), 489
enable_bluetooth()
(digi.xbee.devices.WiFiDevice method), 424
enable_bluetooth()
(digi.xbee.devices.XBeeDevice method), 255
enable_bluetooth()
(digi.xbee.devices.ZigBeeDevice method), 327
enable_logger() (in module digi.xbee.util.utils), 235
enable_logger()
(digi.xbee.models.accesspoint.AccessPoint attribute), 95
enable_logger()
(digi.xbee.models.options.TransmitOptions attribute), 109
enable_logger()
(digi.xbee.models.options.TransmitOptions attribute), 109
encryption_type
(digi.xbee.models.accesspoint.AccessPoint attribute), 95
error
(digi.xbee.packets.devicecloud.FrameErrorPacket attribute), 183
execute_command()
(digi.xbee.devices.AbstractXBeeDevice method), 236
execute_command()
(digi.xbee.devices.AbstractXBeeDevice method), 236
execute_command()
(digi.xbee.devices.CellularDevice method), 366
execute_command()
(digi.xbee.devices.CellularDevice method), 366
execute_command()
(digi.xbee.devices.DigiMeshDevice method), 287
execute_command()
(digi.xbee.devices.DigiPointDevice method), 306
execute_command()
(digi.xbee.devices.DigiMeshDevice method), 288
execute_command()
(digi.xbee.devices.DigiPointDevice method), 306
execute_command()
(digi.xbee.devices.IPDevice method), 349
execute_command()
(digi.xbee.devices.LPWANDevice method), 383
execute_command()
(digi.xbee.devices.LPWANDevice method), 383
execute_command()
(digi.xbee.devices.NBIoTDevice method), 400
execute_command()
(digi.xbee.devices.NBIoTDevice method), 400
execute_command()
(digi.xbee.devices.Raw802Device method), 268
execute_command()
(digi.xbee.devices.RemoteDigiMeshDevice method), 465
execute_command()
(digi.xbee.devices.RemoteDigiPointDevice method), 477
execute_command()
(digi.xbee.devices.RemoteRaw802Device method), 453
execute_command()
(digi.xbee.devices.RemoteXBeeDevice method), 442
execute_command()
(digi.xbee.devices.RemoteZigBeeDevice method), 489
execute_command()
(digi.xbee.devices.WiFiDevice method), 424
execute_command()
(digi.xbee.devices.XBeeDevice method), 255
execute_command()
(digi.xbee.devices.ZigBeeDevice method), 327
enable_logger() (in module digi.xbee.util.utils), 235
ENABLE_MULTICAST (digi.xbee.models.options.TransmitOptions attribute), 109
ENABLE_UNICAST_NACK (digi.xbee.models.options.TransmitOptions attribute), 108
ENABLE_UNICAST_TRACE_ROUTE (digi.xbee.models.options.TransmitOptions attribute), 109
encryption_type (digi.xbee.models.accesspoint.AccessPoint attribute), 95
error (digi.xbee.packets.devicecloud.FrameErrorPacket attribute), 183
execute_command() (digi.xbee.devices.AbstractXBeeDevice method), 236
execute_command() (digi.xbee.devices.AbstractXBeeDevice method), 236
execute_command() (digi.xbee.devices.CellularDevice method), 366
execute_command() (digi.xbee.devices.CellularDevice method), 366
execute_command() (digi.xbee.devices.DigiMeshDevice method), 287
execute_command() (digi.xbee.devices.DigiPointDevice method), 306
execute_command() (digi.xbee.devices.DigiMeshDevice method), 288
execute_command() (digi.xbee.devices.DigiPointDevice method), 306
execute_command() (digi.xbee.devices.IPDevice method), 349
execute_command() (digi.xbee.devices.LPWANDevice method), 383
execute_command() (digi.xbee.devices.LPWANDevice method), 383
execute_command() (digi.xbee.devices.IPDevice method), 349
execute_command() (digi.xbee.devices.LPWANDevice method), 383
execute_command() (digi.xbee.devices.NBIoTDevice method), 401
execute_command() (digi.xbee.devices.Raw802Device method), 269
execute_command() (digi.xbee.devices.RemoteDigiMeshDevice method), 465
execute_command() (digi.xbee.devices.RemoteDigiPointDevice method), 477
execute_command() (digi.xbee.devices.RemoteRaw802Device method), 453
execute_command() (digi.xbee.devices.RemoteXBeeDevice method), 442
execute_command() (digi.xbee.devices.RemoteZigBeeDevice method), 489
execute_command() (digi.xbee.devices.WiFiDevice method), 424
execute_command() (digi.xbee.devices.XBeeDevice method), 255
execute_command() (digi.xbee.devices.ZigBeeDevice method), 327
ExplicitAddressingPacket (class in digi.xbee.packets.common), 166
ExplicitDataReceived (class in digi.xbee.reader), 539
ExplicitRXIndicatorPacket (class in digi.xbee.packets.common), 170
ExplicitXBeeMessage (class in digi.xbee.models.message), 104
extend() (digi.xbee.reader.BlueDataReceived method), 543
extend() (digi.xbee.reader.DataReceived method), 535
extend() (digi.xbee.reader.DeviceDiscovered method), 538
extend() (digi.xbee.reader.DiscoveryProcessFinished method), 538
extend() (digi.xbee.reader.ExplicitDataReceived method), 539
extend() (digi.xbee.reader.IOSampleReceived method), 537
extend() (digi.xbee.reader.IPDataReceived method), 540
extend() (digi.xbee.reader.MicroPythonDataReceived method), 544
extend() (digi.xbee.reader.ModemStatusReceived method), 536
extend() (digi.xbee.reader.PacketReceived method), 534
extend() (digi.xbee.reader.RelayDataReceived method), 542
extend() (digi.xbee.reader.SMSReceived method), 541
extend() (digi.xbee.reader.XBeeEvent method), 533
EXTENDED_TIMEOUT (digi.xbee.models.options.RemoteATCmdOptions attribute), 110

F

file_data (digi.xbee.packets.devicecloud.SendDataRequestPacket attribute), 186
flags (digi.xbee.packets.devicecloud.DeviceRequestPacket attribute), 175
FlowControl (class in digi.xbee.serial), 551
flush() (digi.xbee.reader.XBeeQueue method), 550
flush_queues() (digi.xbee.devices.CellularDevice method), 366
flush_queues() (digi.xbee.devices.DigiMeshDevice method), 288
flush_queues() (digi.xbee.devices.DigiPointDevice method), 307
flush_queues() (digi.xbee.devices.IPDevice method), 349
flush_queues() (digi.xbee.devices.LPWANDevice method), 384
flush_queues() (digi.xbee.devices.NBIoTDevice method), 401
flush_queues() (digi.xbee.devices.Raw802Device method), 269
flush_queues() (digi.xbee.devices.WiFiDevice method), 424
flush_queues() (digi.xbee.devices.XBeeDevice method), 253
flush_queues() (digi.xbee.devices.ZigBeeDevice method), 328
force_disassociate() (digi.xbee.devices.RemoteZigBeeDevice method), 488
force_disassociate() (digi.xbee.devices.ZigBeeDevice method), 323
frame_id (digi.xbee.packets.base.GenericXBeePacket attribute), 127
frame_id (digi.xbee.packets.base.UnknownXBeePacket attribute), 129
frame_id (digi.xbee.packets.base.XBeeAPIPacket attribute), 125
frame_id (digi.xbee.packets.cellular.RXSMSPacket attribute), 132
frame_id (digi.xbee.packets.cellular.TXSMSPacket attribute), 135
frame_id (digi.xbee.packets.common.ATCommPacket attribute), 137
frame_id (digi.xbee.packets.common.ATCommQueuePacket attribute), 140
frame_id (digi.xbee.packets.common.ATCommResponsePacket attribute), 143
frame_id (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 169
frame_id (digi.xbee.packets.common.ExplicitRXIndicatorPacket attribute), 171
frame_id (digi.xbee.packets.common.IODataSampleRxIndicatorPacket attribute), 165
frame_id (digi.xbee.packets.common.ModemStatusPacket attribute), 162
frame_id (digi.xbee.packets.common.ReceivePacket attribute), 146
frame_id (digi.xbee.packets.common.RemoteATCommandPacket attribute), 149
frame_id (digi.xbee.packets.common.RemoteATCommandWifiPacket attribute), 153
frame_id (digi.xbee.packets.common.TransmitPacket attribute), 156
frame_id (digi.xbee.packets.common.TransmitStatusPacket attribute), 159
frame_id (digi.xbee.packets.devicecloud.DeviceRequestPacket attribute), 175
frame_id (digi.xbee.packets.devicecloud.DeviceResponsePacket string()), 178
frame_id (digi.xbee.packets.devicecloud.DeviceResponsePacket StatusPacket), 180
frame_id (digi.xbee.packets.devicecloud.FrameErrorPacket attribute), 183
frame_id (digi.xbee.packets.devicecloud.SendDataRequestPacket attribute), 186
frame_id (digi.xbee.packets.devicecloud.SendDataResponsePacket attribute), 188
frame_id (digi.xbee.packets.network.RXIPv4Packet attribute), 192
frame_id (digi.xbee.packets.network.TXIPv4Packet attribute), 194
frame_id (digi.xbee.packets.raw.RX16Packet attribute), 214
frame_id (digi.xbee.packets.raw.RX64Packet attribute), 209
frame_id (digi.xbee.packets.raw.RX64OPacket attribute), 212
frame_id (digi.xbee.packets.raw.RX64IPacket attribute), 206
frame_id (digi.xbee.packets.raw.TX16Packet attribute), 200
frame_id (digi.xbee.packets.raw.TX46Packet attribute), 132
frame_id (digi.xbee.packets.raw.TX4OPacket attribute), 135
frame_id (digi.xbee.packets.raw.TXStatusPacket attribute), 198
frame_id (digi.xbee.packets.raw.TXStatusPacket attribute), 198
frame_id (digi.xbee.packets.relpy.UserDataRelayOutputPacket attribute), 203
frame_id (digi.xbee.packets.relpy.UserDataRelayPacket attribute), 217
frame_id (digi.xbee.packets.relpy.RemoteATCommandResponsePacket attribute), 220
frame_id (digi.xbee.packets.relpy.RemoteATCommandResponseWifiPacket attribute), 229
frame_id (digi.xbee.packets.relpy.RemoteATCommandRequestPacket attribute), 226
frame_id (digi.xbee.models.address.XBee64BitAddress class method), 102
frame_id (digi.xbee.models.address.XBee64BitAddress class method), 103
frame_id (digi.xbee.models.address.XBee16BitAddress class method), 101
frame_id (digi.xbee.models.address.XBeeIMEIAddress class method), 103
frame_id (digi.xbee.reader.XBeeQueue method), 550
frame_id (digi.xbee.reader.XBeeQueue method), 548
frame_id (digi.xbee.reader.XBeeQueue method), 548
get() (digi.xbee.devices.AbstractXBeeDevice method), 126
get_16_bit_addr() (digi.xbee.devices.CellularDevice method), 238
get_16_bit_addr() (digi.xbee.devices.DigiMeshDevice method), 366
get_16_bit_addr() (digi.xbee.devices.DigiPointDevice method), 288
get_16_bit_addr() (digi.xbee.devices.IPDevice method), 307
get_16_bit_addr() (digi.xbee.devices.LPWANDevice method), 346
get_16_bit_addr() (digi.xbee.devices.NBIoTDevice method), 384
get_16_bit_addr() (digi.xbee.devices.NBIoTDevice method), 401
get_16_bit_addr() (digi.xbee.devices.Raw802Device method), 269

Index 573
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Device Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_16bit_addr()</td>
<td>RemoteDigiMeshDevice</td>
<td>465</td>
</tr>
<tr>
<td>get_16bit_addr()</td>
<td>RemoteDigiPointDevice</td>
<td>477</td>
</tr>
<tr>
<td>get_16bit_addr()</td>
<td>RemoteRaw802Device</td>
<td>453</td>
</tr>
<tr>
<td>get_16bit_addr()</td>
<td>RemoteXBeeDevice</td>
<td>442</td>
</tr>
<tr>
<td>get_16bit_addr()</td>
<td>RemoteZigBeeDevice</td>
<td>489</td>
</tr>
<tr>
<td>get_16bit_addr()</td>
<td>WiFiDevice</td>
<td>255</td>
</tr>
<tr>
<td>get_16bit_addr()</td>
<td>XBeeDevice</td>
<td>328</td>
</tr>
<tr>
<td>get_16bit_addr()</td>
<td>ZigBeeDevice</td>
<td>574</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>AbstractXBeeDevice</td>
<td>239</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>CellularDevice</td>
<td>363</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>DigiMeshDevice</td>
<td>288</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>DigiPointDevice</td>
<td>307</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>IPDevice</td>
<td>269</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>LPWANDevice</td>
<td>349</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>NBIoTDevice</td>
<td>384</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>Raw802Device</td>
<td>454</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>RemoteDigiMeshDevice</td>
<td>466</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>RemoteDigiPointDevice</td>
<td>478</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>RemoteRaw802Device</td>
<td>443</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>RemoteXBeeDevice</td>
<td>307</td>
</tr>
<tr>
<td>get_64bit_addr()</td>
<td>RemoteZigBeeDevice</td>
<td>328</td>
</tr>
<tr>
<td>get_access_point()</td>
<td>WiFiDevice</td>
<td>256</td>
</tr>
<tr>
<td>get_access_point_timeout()</td>
<td>WiFiDevice</td>
<td>367</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>XBeeDevice</td>
<td>322</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>CellularDevice</td>
<td>266</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>RemoteDigiMeshDevice</td>
<td>490</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>RemoteDigiPointDevice</td>
<td>452</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>RemoteRaw802Device</td>
<td>443</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>RemoteZigBeeDevice</td>
<td>328</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>WiFiDevice</td>
<td>256</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>XBeeDevice</td>
<td>322</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>ZigBeeDevice</td>
<td>322</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>IPDevice</td>
<td>256</td>
</tr>
<tr>
<td>get_adc_value()</td>
<td>CellularDevice</td>
<td>322</td>
</tr>
<tr>
<td>get_ai_status()</td>
<td>RemoteRaw802Device</td>
<td>266</td>
</tr>
<tr>
<td>get_api_output_mode()</td>
<td>RemoteDigiMeshDevice</td>
<td>490</td>
</tr>
<tr>
<td>get_api_output_mode()</td>
<td>WiFiDevice</td>
<td>245</td>
</tr>
<tr>
<td>get_api_output_mode()</td>
<td>CellularDevice</td>
<td>256</td>
</tr>
<tr>
<td>get_api_output_mode()</td>
<td>ZigBeeDevice</td>
<td>289</td>
</tr>
<tr>
<td>get_api_output_mode()</td>
<td>RemoteDigiPointDevice</td>
<td>490</td>
</tr>
<tr>
<td>get_analog_value()</td>
<td>RemoteDigiMeshDevice</td>
<td>531</td>
</tr>
<tr>
<td>get_analog_value()</td>
<td>WiFiDevice</td>
<td>245</td>
</tr>
<tr>
<td>get_analog_value()</td>
<td>CellularDevice</td>
<td>289</td>
</tr>
<tr>
<td>get_analog_value()</td>
<td>ZigBeeDevice</td>
<td>289</td>
</tr>
<tr>
<td>get_analog_value()</td>
<td>RemoteDigiPointDevice</td>
<td>289</td>
</tr>
<tr>
<td>get_api_output_mode()</td>
<td>RemoteDigiMeshDevice</td>
<td>242</td>
</tr>
</tbody>
</table>
get_api_output_mode()
(digi.xbee.devices.LPWANDevice method), 270
get_api_output_mode()
(digi.xbee.devices.NBIoTDevice method), 467
get_api_output_mode()
(digi.xbee.devices.Raw802Device method), 478
get_api_output_mode()
(digi.xbee.devices.RemoteDigiMeshDevice method), 478
get_api_output_mode()
(digi.xbee.devices.RemoteRaw802Device method), 455
get_api_output_mode()
(digi.xbee.devices.RemoteXBeeDevice method), 444
get_api_output_mode()
(digi.xbee.devices.RemoteZigBeeDevice method), 491
get_api_output_mode()
(digi.xbee.devices.ZigBeeDevice method), 329
get_by_id()
(digi.xbee.reader.XBeeQueue method), 550
get_by_ip()
(digi.xbee.reader.XBeeQueue method), 549
get_by_remote()
(digi.xbee.reader.XBeeQueue method), 549
get_cellular_ai_status()
(digi.xbee.devices.CellularDevice method), 362
get_cellular_ai_status()
(digi.xbee.devices.LPWANDevice method), 385
get_cellular_ai_status()
(digi.xbee.devices.NBIoTDevice method), 402
get_checksum()
(digi.xbee.packets.base.GenericXBeePacket method), 127
get_checksum()
(digi.xbee.packets.base.UnknownXBeePacket method), 129
get_checksum()
(digi.xbee.packets.base.XBeeAPIPacket method), 125
get_checksum()
(digi.xbee.packets.base.XBeePacket method), 122
get_checksum()
(digi.xbee.packets.cellular.RXSMSPacket method), 132
get_checksum()
(digi.xbee.packets.cellular.TXSMSPacket method), 135
get_checksum()
(digi.xbee.packets.common.ATCommPacket method), 138
get_checksum()
(digi.xbee.packets.common.ATCommQueuePacket method), 138

Index 575
XBee Python Library Documentation, Release 1.2.0

get_checksum() (digi.xbee.packets.common.ATCommResponsePacket method), 143
get_checksum() (digi.xbee.packets.common.ExplicitAddressingPacket method), 169
get_checksum() (digi.xbee.packets.common.ExplicitRXIndicatorPacket method), 171
get_checksum() (digi.xbee.packets.common.IODataSampleRxIndicatorPacket method), 165
get_checksum() (digi.xbee.packets.common.ModemStatusPacket method), 162
get_checksum() (digi.xbee.packets.common.ReceivePacket method), 146
get_checksum() (digi.xbee.packets.common.RemoteATCommandPacket method), 149
get_checksum() (digi.xbee.packets.common.RemoteATCommandResponsePacket method), 153
get_checksum() (digi.xbee.packets.common.TransmitPacket method), 156
get_checksum() (digi.xbee.packets.common.TransmitStatusPacket method), 159
get_checksum() (digi.xbee.packets.devicecloud.DeviceRequestPacket method), 175
get_checksum() (digi.xbee.packets.devicecloud.DeviceResponsePacket method), 178
get_checksum() (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket method), 181
get_checksum() (digi.xbee.packets.devicecloud.FrameErrorPacket method), 183
get_checksum() (digi.xbee.packets.devicecloud.SendDataRequestPacket method), 186
get_checksum() (digi.xbee.packets.devicecloud.SendDataResponsePacket method), 188
get_checksum() (digi.xbee.packets.network.RXIPv4Packet method), 192
get_checksum() (digi.xbee.packets.network.TXIPv4Packet method), 195
get_checksum() (digi.xbee.packets.raw.RX16IOPacket method), 214
get_checksum() (digi.xbee.packets.raw.RX16Packet method), 209
get_checksum() (digi.xbee.packets.raw.RX64IOPacket method), 212
get_checksum() (digi.xbee.packets.raw.RX64Packet method), 206
get_checksum() (digi.xbee.packets.raw.TX16Packet method), 201
get_checksum() (digi.xbee.packets.raw.TX64Packet method), 198
get_checksum() (digi.xbee.packets.raw.TXStatusPacket method), 198
get_checksum() (digi.xbee.packets.relax.UserDataRelayOutputPacket method), 220
get_checksum() (digi.xbee.packets.relax.UserDataRelayPacket method), 217
get_checksum() (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 223
get_checksum() (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket method), 226
get_checksum() (digi.xbee.packets.wifi.RemoteATCommandWifiPacket method), 229
get_checksum() (digi.xbee.packets.wifi.RemoteATCommandWifiPacket method), 232
get_checksum() (digi.xbee.packets.wifi.RemoteATCommandWifiPacket method), 233
get_checksum() (digi.xbee.devices.AbstractXBeeDevice method), 239
get_current_frame_id() (digi.xbee.devices.CellularDevice method), 368
get_current_frame_id() (digi.xbee.devices.DigiMeshDevice method), 309
get_current_frame_id() (digi.xbee.devices.DigiPointDevice method), 309
get_current_frame_id() (digi.xbee.devices.IPDevice method), 351
get_current_frame_id() (digi.xbee.devices.LPWANDevice method), 402
get_current_frame_id() (digi.xbee.devices.NBIoTDevice method), 422
get_current_frame_id() (digi.xbee.devices.RAWDevice method), 455
get_current_frame_id() (digi.xbee.devices.RemoteDigiMeshDevice method), 467
get_current_frame_id() (digi.xbee.devices.RemoteDigiMeshDevice method), 479
get_current_frame_id() (digi.xbee.devices.RemoteRaw802Device method), 491
get_current_frame_id() (digi.xbee.devices.RemoteXBeeDevice method), 491
get_current_frame_id() (digi.xbee.devices.RemoteZigBeeDevice method), 503
get_current_frame_id() (digi.xbee.devices.XBeeDevice method), 517
get_current_frame_id() (digi.xbee.devices.ZigBeeDevice method), 531
get_data_queue() (digi.xbee.reader.PacketListener method), 576
get_dest_address() (digi.xbee.devices.AbstractXBeeDevice method), 240

get_dest_address() (digi.xbee.devices.CellularDevice method), 368

get_dest_address() (digi.xbee.devices.DigiMeshDevice method), 290

get_dest_address() (digi.xbee.devices.DigiPointDevice method), 309

get_dest_address() (digi.xbee.devices.IPDevice method), 346

get_dest_address() (digi.xbee.devices.LPWANDevice method), 385

get_dest_address() (digi.xbee.devices.NBIoTDevice method), 403

get_dest_address() (digi.xbee.devices.Raw802Device method), 271

get_device_by_16() (digi.xbee.devices.DigiMeshNetwork method), 518

get_device_by_16() (digi.xbee.devices.DigiPointNetwork method), 523

get_device_by_16() (digi.xbee.devices.Raw802Network method), 513

get_device_by_16() (digi.xbee.devices.XBeeNetwork method), 504

get_device_by_16() (digi.xbee.devices.ZigBeeNetwork method), 507

get_device_by_64() (digi.xbee.devices.DigiMeshNetwork method), 518

get_device_by_64() (digi.xbee.devices.DigiPointNetwork method), 523

get_device_by_64() (digi.xbee.devices.Raw802Network method), 513

get_device_by_64() (digi.xbee.devices.XBeeNetwork method), 503

get_device_by_64() (digi.xbee.devices.ZigBeeNetwork method), 508

get_device_by_node_id() (digi.xbee.devices.DigiMeshNetwork method), 518

get_device_by_node_id() (digi.xbee.devices.DigiPointNetwork method), 523

get_device_by_node_id() (digi.xbee.devices.XBeeNetwork method), 504

get_device_by_node_id() (digi.xbee.devices.ZigBeeNetwork method), 508

get_ip_addr() (digi.xbee.devices.AbstractXBeeDevice method), 385

get_ip_addr() (digi.xbee.devices.NBIoTDevice method), 403

get_ip_addr() (digi.xbee.devices.WiFiDevice method), 426

get_ip_addr() (digi.xbee.devices.XBeeDevice method), 257

get_ip_addr() (digi.xbee.devices.ZigBeeDevice method), 329

get_ip_addr() (digi.xbee.devices.CellularDevice method), 368

get_ip_addr() (digi.xbee.devices.IPDevice method), 343

get_ip_addr() (digi.xbee.devices.LPWANDevice method), 426
get_devices() (digi.xbee.devices.AbstractXBeeDevice method), 501
get_devices() (digi.xbee.devices.CellularDevice method), 290
get_devices() (digi.xbee.devices.DigiMeshDevice method), 244
get_devices() (digi.xbee.devices.DigiPointDevice method), 309
get_devices() (digi.xbee.devices.DigiPointDevice method), 368
get_devices() (digi.xbee.devices.IPDevice method), 290
get_devices() (digi.xbee.devices.LPWANDevice method), 386
get_devices() (digi.xbee.devices.NBLoTDevice method), 403
get_devices() (digi.xbee.devices.RemoteDigiMeshDevice method), 479
get_devices() (digi.xbee.devices.RemoteDigiPointDevice method), 503
get_devices() (digi.xbee.devices.RemoteRaw802Device method), 455
getDevices() (digi.xbee.devices.Raw802Network method), 513
getDevices() (digi.xbee.devices.XBeeNetwork method), 508
get_discovery_callbacks() (digi.xbee.devices.AbstractXBeeDevice method), 519
get_discovery_callbacks() (digi.xbee.devices.DigiMeshNetwork method), 519
get_discovery_callbacks() (digi.xbee.devices.DigiPointNetwork method), 524
get_discovery_callbacks() (digi.xbee.devices.XBeeNetwork method), 502
get_discovery_options() (digi.xbee.devices.AbstractXBeeDevice method), 508
get_discovery_options() (digi.xbee.devices.DigiMeshNetwork method), 519
get_discovery_options() (digi.xbee.devices.DigiPointNetwork method), 524
get_discovery_options() (digi.xbee.devices.XBeeNetwork method), 502
get_discovery_timeout() (digi.xbee.devices.AbstractXBeeDevice method), 508
get_discovery_timeout() (digi.xbee.devices.DigiMeshNetwork method), 519
get_discovery_timeout() (digi.xbee.devices.DigiPointNetwork method), 524
get_discovery_timeout() (digi.xbee.devices.XBeeNetwork method), 503
get_discovery_timeout() (digi.xbee.devices.ZigBeeNetwork method), 508
get_dns_address() (digi.xbee.devices.XBeeDevice method), 257
get_explicit_queue() (digi.xbee.reader.PacketListener method), 545
get_firmware_version() (digi.xbee.devices.AbstractXBeeDevice method), 238
get_firmware_version() (digi.xbee.devices.CellularDevice method), 369
get_firmware_version() (digi.xbee.devices.DigiMeshDevice method), 290
get_firmware_version() (digi.xbee.devices.DigiPointDevice method), 309
get_firmware_version() (digi.xbee.devices.IPDevice method), 351
get_firmware_version() (digi.xbee.devices.LPWANDevice method), 386
get_firmware_version() (digi.xbee.devices.NBLoTDevice method), 403
get_firmware_version() (digi.xbee.devices.RemoteDigiMeshDevice method), 514
get_firmware_version() (digi.xbee.devices.RemoteDigiPointDevice method), 519
get_firmware_version() (digi.xbee.devices.RemoteRaw802Device method), 524
get_firmware_version() (digi.xbee.devices.XBeeNetwork method), 503
get_firmware_version() (digi.xbee.devices.ZigBeeNetwork method), 508
get_discovery_callbacks() (digi.xbee.devices.AbstractXBeeDevice method), 502
get_discovery_callbacks() (digi.xbee.devices.DigiMeshNetwork method), 519
get_discovery_callbacks() (digi.xbee.devices.DigiPointNetwork method), 524
get_discovery_callbacks() (digi.xbee.devices.Raw802Network method), 513
get_discovery_callbacks() (digi.xbee.devices.XBeeNetwork method), 508
get_firmware_version() (digi.xbee.devices.XBeeDevice method), 257
get_firmware_version()
(digi.xbee.devices.NBIoTDevice method), 403
get_firmware_version()
(digi.xbee.devices.Raw802Device method), 271
get_firmware_version()
(digi.xbee.devices.RemoteDigiMeshDevice method), 468
get_firmware_version()
(digi.xbee.devices.RemoteDigiPointDevice method), 480
get_firmware_version()
(digi.xbee.devices.RemoteRaw802Device method), 456
get_firmware_version()
(digi.xbee.devices.RemoteXBeeDevice method), 445
get_firmware_version()
(digi.xbee.devices.RemoteZigBeeDevice method), 492
get_firmware_version()
(digi.xbee.devices.WiFiDevice method), 427
get_firmware_version()
(digi.xbee.devices.XBeeDevice method), 258
get_firmware_version()
(digi.xbee.devices.ZigBeeDevice method), 330
get_frame_spec_data()
(digi.xbee.packets.base.GenericXBeePacket method), 127
get_frame_spec_data()
(digi.xbee.packets.base.UnknownXBeePacket method), 129
get_frame_spec_data()
(digi.xbee.packets.base.XBeeAPIPacket method), 124
get_frame_spec_data()
(digi.xbee.packets.base.XBeePacket method), 123
get_frame_spec_data()
(digi.xbee.packets.cellular.RXSMSPacket method), 132
get_frame_spec_data()
(digi.xbee.packets.cellular.TXSMSPacket method), 135
get_frame_spec_data()
(digi.xbee.packets.common.ATCommResponsePacket method), 144
get_frame_spec_data()
(digi.xbee.packets.common.ExplicitAddressingPacket method), 169
get_frame_spec_data()
(digi.xbee.packets.common.ExplicitRXIndicatorPacket method), 171
get_frame_spec_data()
(digi.xbee.packets.common.IODataSampleRxIndicatorPacket method), 165
get_frame_spec_data()
(digi.xbee.packets.common.ModemStatusPacket method), 162
get_frame_spec_data()
(digi.xbee.packets.common.ReceivePacket method), 147
get_frame_spec_data()
(digi.xbee.packets.common.RemoteATCommandPacket method), 150
get_frame_spec_data()
(digi.xbee.packets.common.RemoteATCommandResponsePacket method), 153
get_frame_spec_data()
(digi.xbee.packets.common.TransmitPacket method), 156
get_frame_spec_data()
(digi.xbee.packets.common.TransmitStatusPacket method), 159
get_frame_spec_data()
(digi.xbee.packets.devicecloud.DeviceRequestPacket method), 175
get_frame_spec_data()
(digi.xbee.packets.devicecloud.DeviceResponsePacket method), 178
get_frame_spec_data()
(digi.xbee.packets.devicecloud.DeviceResponseStatusPacket method), 181
get_frame_spec_data()
(digi.xbee.packets.devicecloud.FrameErrorPacket method), 183
get_frame_spec_data()
(digi.xbee.packets.devicecloud.SendDataRequestPacket method), 186
get_frame_spec_data()
(digi.xbee.packets.devicecloud.SendDataResponsePacket method), 188
get_frame_spec_data()
(digi.xbee.packets.network.RXIPv4Packet method), 192
get_frame_spec_data()
(digi.xbee.packets.network.TXIPv4Packet method), 195
get_frame_spec_data()
get_frame_type()
digi.xbee.packets.wifi.RemoteATCommandWifiPacket
209
get_frame_spec_data()
digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket
212
get_frame_type()
digi.xbee.packets.raw.RX64Packet
206
get_frame_spec_data()
digi.xbee.packets.raw.TX16Packet
201
get_frame_spec_data()
digi.xbee.packets.raw.TX64Packet
198
get_frame_spec_data()
digi.xbee.packets.common.ExplicitRXIndicatorPacket
169
digi.xbee.packets.common.ExplicitAddressingPacket
144
digi.xbee.packets.common.ATCommResponsePacket
141
digi.xbee.packets.common.ATCommQueuePacket
138
digi.xbee.packets.common.ATCommPacket
124
digi.xbee.packets.cellular.TXSMSPacket
129
digi.xbee.packets.cellular.RXSMSPacket
127
digi.xbee.packets.base.XBeeAPIPacket
122
digi.xbee.packets.base.UnknownXBeePacket
115
digi.xbee.packets.base.GenericXBeePacket
108
(digi.xbee.packets.raw.RX16IOPacket method), 172
get_frame_type()
digi.xbee.packets.common.IODataSampleRxIndicatorPacket
165
digi.xbee.packets.common.ModemStatusPacket
162
get_frame_type()
digi.xbee.packets.common.ReceivePacket
147
digi.xbee.packets.common.RemoteATCommandPacket
150
digi.xbee.packets.common.RemoteATCommandResponsePacket
153
digi.xbee.packets.common.TransmitPacket
156
digi.xbee.packets.common.TransmitStatusPacket
160
digi.xbee.packets.devicecloud.DeviceRequestPacket
176
digi.xbee.packets.devicecloud.DeviceResponsePacket
178
digi.xbee.packets.devicecloud.DeviceResponseStatusPacket
181
digi.xbee.packets.devicecloud.DeviceResponseStatusPacket
184
digi.xbee.packets.devicecloud.SendDataRequestPacket
186
digi.xbee.packets.devicecloud.SendDataResponsePacket
188
digi.xbee.packets.network.RXIPv4Packet
192
digi.xbee.packets.network.TXIPv4Packet
195
digi.xbee.packets.raw.RX16IOPacket
198
get_frame_type()
digi.xbee.packets.raw.RX64Packet
201
get_frame_type()
digi.xbee.packets.raw.TX64Packet
204
get_frame_type()
digi.xbee.packets.raw.TX16Packet
200
get_frame_type()
digi.xbee.packets.raw.TX64IOPacket
195
get_frame_type()
digi.xbee.packets.raw.RX16Packet
192
get_frame_type()
digi.xbee.packets.raw.RX16IOPacket
188
get_frame_type()
digi.xbee.packets.raw.TX64Packet
181
get_frame_type()
digi.xbee.packets.raw.TX16Packet
174
get_frame_type()
digi.xbee.packets.raw.TX64IOPacket
167
get_frame_type()
digi.xbee.packets.raw.TX16Packet
160
get_frame_type()
digi.xbee.packets.raw.TX64Packet
153
get_frame_type()
digi.xbee.packets.raw.TX16Packet
146
get_frame_type_value()
(digi.xbee.packets.base.GenericXBeePacket method), 127
get_frame_type_value()
(digi.xbee.packets.base.UnknownXBeePacket method), 130
get_frame_type_value()
(digi.xbee.packets.base.XBeeAPIPacket method), 124
get_frame_type_value()
(digi.xbee.packets.cellular.RXSMSPacket method), 133
get_frame_type_value()
(digi.xbee.packets.cellular.TXSMSPacket method), 135
get_frame_type_value()
(digi.xbee.packets.common.ATCommPacket method), 138
get_frame_type_value()
(digi.xbee.packets.common.ATCommQueuePacket method), 141
get_frame_type_value()
(digi.xbee.packets.common.ATCommResponsePacket method), 144
get_frame_type_value()
(digi.xbee.packets.common.ExplicitAddressingPacket method), 169
get_frame_type_value()
(digi.xbee.packets.common.ExplicitRXIndicatorPacket method), 172
get_frame_type_value()
(digi.xbee.packets.common.IODataSampleRxIndicatorPacket method), 166
get_frame_type_value()
(digi.xbee.packets.common.ModemStatusPacket method), 162
get_frame_type_value()
(digi.xbee.packets.common.ReceivePacket method), 147
get_frame_type_value()
(digi.xbee.packets.common.RemoteATCommandPacket method), 150
get_frame_type_value()
(digi.xbee.packets.common.RemoteATCommandResponsePacket method), 153
get_frame_type_value()
(digi.xbee.packets.common.TransmitPacket method), 157
get_frame_type_value()
(digi.xbee.packets.common.TransmitStatusPacket method), 160
get_frame_type_value()
(digi.xbee.packets.devicecloud.DeviceRequestPacket method), 176
get_frame_type_value()
(digi.xbee.packets.devicecloud.DeviceResponsePacket method), 179
get_frame_type_value()
(digi.xbee.packets.devicecloud.DeviceResponseStatusPacket method), 181
get_frame_type_value()
(digi.xbee.packets.devicecloud.FrameErrorPacket method), 184
get_frame_type_value()
(digi.xbee.packets.devicecloud.SendDataRequestPacket method), 187
get_frame_type_value()
(digi.xbee.packets.devicecloud.SendDataResponsePacket method), 189
get_frame_type_value()
(digi.xbee.packets.devicecloud.SendDataResponseStatusPacket method), 192
get_frame_type_value()
(digi.xbee.packets.devicecloud.TXIPv4Packet method), 195
get_frame_type_value()
(digi.xbee.packets.raw.RX16IOPacket method), 214
get_frame_type_value()
(digi.xbee.packets.raw.RX16Packet method), 210
get_frame_type_value()
(digi.xbee.packets.raw.RX64IOPacket method), 212
get_frame_type_value()
(digi.xbee.packets.raw.RX64Packet method), 207
get_frame_type_value()
(digi.xbee.packets.raw.TX16Packet method), 201
get_frame_type_value()
(digi.xbee.packets.raw.TX64Packet method), 198
get_frame_type_value()
(digi.xbee.packets.raw.TXStatusPacket method), 204
get_frame_type_value()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 221
get_frame_type_value()
(digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket method), 224
get_frame_type_value()
(digi.xbee.packets.wifi.RemoteATCommandWifiPacket
method), 230
get_gateway_address()
(digi.xbee.devices.WiFiDevice
method), 421
get_hardware_version()
(digi.xbee.devices.AbstractXBeeDevice
method), 238
get_hardware_version()
(digi.xbee.devices.CellularDevice
method), 369
get_hardware_version()
(digi.xbee.devices.DigiMeshDevice
method), 291
get_hardware_version()
(digi.xbee.devices.DigiPointDevice
method), 310
get_hardware_version()
(digi.xbee.devices.IPDevice
method), 352
get_hardware_version()
(digi.xbee.devices.LPWANDevice
method), 387
get_hardware_version()
(digi.xbee.devices.NBIoTDevice
method), 404
get_hardware_version()
(digi.xbee.devices.Raw802Device
method), 272
get_hardware_version()
(digi.xbee.devices.RemoteDigiMeshDevice
method), 468
get_hardware_version()
(digi.xbee.devices.RemoteDigiPointDevice
method), 480
get_hardware_version()
(digi.xbee.devices.RemoteRaw802Device
method), 456
get_hardware_version()
(digi.xbee.devices.RemoteXBeeDevice
method), 445
get_hardware_version()
(digi.xbee.devices.RemoteZigBeeDevice
method), 492
get_hardware_version()
(digi.xbee.devices.WiFiDevice
method), 427
get_hardware_version()
(digi.xbee.devices.XBeeDevice
method), 258
get_hardware_version()
(digi.xbee.devices.ZigBeeDevice
method), 330
get_hsb()
(digi.xbee.models.address.XBee16BitAddress
method), 102
get_imei_addr()
(digi.xbee.devices.CellularDevice
method), 362
get_imei_addr()
(digi.xbee.devices.LPWANDevice
method), 386
get_imei_addr()
(digi.xbee.devices.NBIoTDevice
method), 404
get_io_configuration()
(digi.xbee.devices.AbstractXBeeDevice
method), 241
get_io_configuration()
(digi.xbee.devices.CellularDevice
method), 369
get_io_configuration()
(digi.xbee.devices.DigiMeshDevice
method), 291
get_io_configuration()
(digi.xbee.devices.DigiPointDevice
method), 310
get_io_configuration()
(digi.xbee.devices.IPDevice
method), 352
get_io_configuration()
(digi.xbee.devices.LPWANDevice
method), 387
get_io_configuration()
(digi.xbee.devices.NBIoTDevice
method), 404
get_io_configuration()
(digi.xbee.devices.Raw802Device
method), 272
get_io_configuration()
(digi.xbee.devices.RemoteDigiMeshDevice
method), 468
get_io_configuration()
(digi.xbee.devices.RemoteDigiPointDevice
method), 480
get_io_configuration()
(digi.xbee.devices.RemoteRaw802Device
method), 456
get_io_configuration()
(digi.xbee.devices.RemoteXBeeDevice
method), 445
get_io_configuration()
(digi.xbee.devices.RemoteZigBeeDevice
method), 492
get_io_configuration()
(digi.xbee.devices.WiFiDevice
method), 427
get_io_configuration()
(digi.xbee.devices.XBeeDevice
method), 258
get_io_configuration()
(digi.xbee.devices.ZigBeeDevice
method), 330
get_io_sampling_rate()
(digi.xbee.devices.AbstractXBeeDevice
method), 445
get_io_sampling_rate() (digi.xbee.devices.CellularDevice method), 291
get_io_sampling_rate() (digi.xbee.devices.DigiMeshDevice method), 310
get_io_sampling_rate() (digi.xbee.devices.DigiPointDevice method), 352
get_io_sampling_rate() (digi.xbee.devices.IPDevice method), 387
get_io_sampling_rate() (digi.xbee.devices.LPWANDevice method), 404
get_io_sampling_rate() (digi.xbee.devices.NBIoTDevice method), 428
get_io_sampling_rate() (digi.xbee.devices.Raw802Device method), 428
get_io_sampling_rate() (digi.xbee.devices.RemoteDigiMeshDevice method), 469
get_io_sampling_rate() (digi.xbee.devices.RemoteDigiPointDevice method), 481
get_io_sampling_rate() (digi.xbee.devices.RemoteRaw802Device method), 457
get_io_sampling_rate() (digi.xbee.devices.RemoteXBeeDevice method), 441
get_io_sampling_rate() (digi.xbee.devices.RemoteZigBeeDevice method), 493
get_ip_addressing_mode() (digi.xbee.devices.WiFiDevice method), 419
get_ip_queue() (digi.xbee.reader.PacketListener method), 545
get_local_xbee_device() (digi.xbee.devices.RemoteDigiMeshDevice method), 369
get_local_xbee_device() (digi.xbee.devices.RemoteDigiPointDevice method), 343
get_local_xbee_device() (digi.xbee.devices.RemoteRaw802Device method), 346
get_local_xbee_device() (digi.xbee.devices.RemoteXBeeDevice method), 387
get_local_xbee_device() (digi.xbee.devices.RemoteZigBeeDevice method), 420
get_network() (digi.xbee.devices.CellularDevice method), 363
get_network() (digi.xbee.devices.DigiMeshDevice method), 291
get_network() (digi.xbee.devices.DigiPointDevice method), 310
get_network() (digi.xbee.devices.IPDevice method), 352
get_network() (digi.xbee.devices.LPWANDevice method), 387
get_network() (digi.xbee.devices.NBIoTDevice method), 404
get_network() (digi.xbee.devices.Raw802Device method), 272
get_network() (digi.xbee.devices.XBeeDevice method), 259
get_network() (digi.xbee.devices.ZigBeeDevice method), 331
get_ip_addr() (digi.xbee.devices.CellularDevice method), 369
get_ip_addr() (digi.xbee.devices.IPDevice method), 387
get_ip_addr() (digi.xbee.devices.LPWANDevice method), 387
get_ip_addr() (digi.xbee.devices.NBIoTDevice method), 404
get_ip_addr() (digi.xbee.devices.WiFiDevice method), 428
get_mask_address() (digi.xbee.devices.WiFiDevice method), 419
get_next_frame_id() (digi.xbee.devices.CellularDevice method), 370
get_next_frame_id() (digi.xbee.devices.DigiMeshDevice method), 284
get_next_frame_id() (digi.xbee.devices.DigiPointDevice method), 303
get_next_frame_id() (digi.xbee.devices.IPDevice method), 352
get_next_frame_id() (digi.xbee.devices.LPWANDevice method), 387
get_next_frame_id() (digi.xbee.devices.NBIoTDevice method), 405
get_next_frame_id() (digi.xbee.devices.Raw802Device method), 265
get_next_frame_id() (digi.xbee.devices.WiFiDevice method), 428
get_next_frame_id() (digi.xbee.devices.XBeeDevice method), 254
get_next_frame_id() (digi.xbee.devices.ZigBeeDevice method), 322
get_next_frame_id() (digi.xbee.devices.CellularDevice method), 370
get_next_frame_id() (digi.xbee.devices.DigiMeshDevice method), 291
get_next_frame_id() (digi.xbee.devices.DigiPointDevice method), 310
get_next_frame_id() (digi.xbee.devices.IPDevice method), 352
get_next_frame_id() (digi.xbee.devices.LPWANDevice method), 387
get_next_frame_id() (digi.xbee.devices.NBIoTDevice method), 405
get_next_frame_id() (digi.xbee.devices.Raw802Device method), 272
get_next_frame_id() (digi.xbee.devices.WiFiDevice method), 428
get_next_frame_id() (digi.xbee.devices.XBeeDevice method), 254
get_next_frame_id() (digi.xbee.devices.ZigBeeDevice method), 331
get_node_id() (digi.xbee.devices.AbstractXBeeDevice method), 238
get_node_id() (digi.xbee.devices.CellularDevice method), 364
get_node_id() (digi.xbee.devices.DigiMeshDevice method), 291
get_node_id() (digi.xbee.devices.DigiPointDevice method), 310
get_node_id() (digi.xbee.devices.IPDevice method), 346
get_node_id() (digi.xbee.devices.LPWANDevice method), 387
get_node_id() (digi.xbee.devices.NBIoTDevice method), 405
get_node_id() (digi.xbee.devices.Raw802Device method), 273
get_node_id() (digi.xbee.devices.RemoteDigiMeshDevice method), 469
get_node_id() (digi.xbee.devices.RemoteDigiPointDevice method), 481
get_node_id() (digi.xbee.devices.RemoteRaw802Device method), 457
get_node_id() (digi.xbee.devices.RemoteXBeeDevice method), 446
get_node_id() (digi.xbee.devices.RemoteZigBeeDevice method), 493
get_node_id() (digi.xbee.devices.XBeeDevice method), 259
get_node_id() (digi.xbee.devices.ZigBeeDevice method), 331
get_nowait() (digi.xbee.reader.XBeeQueue method), 550
get_number_devices() (digi.xbee.devices.DigiMeshNetwork method), 519
get_number_devices() (digi.xbee.devices.DigiPointNetwork method), 524
get_number_devices() (digi.xbee.devices.Raw802Network method), 514
get_number_devices() (digi.xbee.devices.XBeeNetwork method), 501
get_number_devices() (digi.xbee.devices.ZigBeeNetwork method), 509
get_pan_id() (digi.xbee.devices.AbstractXBeeDevice method), 240
get_pan_id() (digi.xbee.devices.CellularDevice method), 370
get_pan_id() (digi.xbee.devices.DigiMeshDevice method), 292
get_pan_id() (digi.xbee.devices.DigiPointDevice method), 311
get_pan_id() (digi.xbee.devices.IPDevice method), 346
get_pan_id() (digi.xbee.devices.LPWANDevice method), 387
get_pan_id() (digi.xbee.devices.NBIoTDevice method), 405
get_pan_id() (digi.xbee.devices.Raw802Device method), 273
get_pan_id() (digi.xbee.devices.RemoteDigiMeshDevice method), 469
get_pan_id() (digi.xbee.devices.RemoteDigiPointDevice method), 481
get_pan_id() (digi.xbee.devices.RemoteRaw802Device method), 457
get_pan_id() (digi.xbee.devices.RemoteXBeeDevice method), 446
get_pan_id() (digi.xbee.devices.RemoteZigBeeDevice method), 493
get_pan_id() (digi.xbee.devices.XBeeDevice method), 259
get_pan_id() (digi.xbee.devices.ZigBeeDevice method), 331
get_parameter() (digi.xbee.devices.AbstractXBeeDevice method), 236
get_parameter() (digi.xbee.devices.CellularDevice method), 370
get_parameter() (digi.xbee.devices.DigiMeshDevice method), 292
get_parameter() (digi.xbee.devices.DigiPointDevice method), 311
get_parameter() (digi.xbee.devices.IPDevice method), 352
get_parameter() (digi.xbee.devices.LPWANDevice method), 387
get_parameter() (digi.xbee.devices.NBLoTDevice method), 405
get_parameter() (digi.xbee.devices.Raw802Device method), 273
get_parameter() (digi.xbee.devices.RemoteDigiMeshDevice method), 469
get_parameter() (digi.xbee.devices.RemoteDigiPointDevice method), 481
get_parameter() (digi.xbee.devices.RemoteRaw802Device method), 457
get_parameter() (digi.xbee.devices.RemoteXBeeDevice method), 446
get_parameter() (digi.xbee.devices.RemoteZigBeeDevice method), 493
get_power_level() (digi.xbee.devices.RemoteDigiMeshDevice method), 469
get_power_level() (digi.xbee.devices.RemoteDigiPointDevice method), 481
get_power_level() (digi.xbee.devices.RemoteRaw802Device method), 457
get_power_level() (digi.xbee.devices.RemoteXBeeDevice method), 446
get_power_level() (digi.xbee.devices.RemoteZigBeeDevice method), 493
get_power_level() (digi.xbee.devices.WiFiDevice method), 429
get_power_level() (digi.xbee.devices.XBeeDevice method), 259
get_power_level() (digi.xbee.devices.ZigBeeDevice method), 331
get_protocol() (digi.xbee.devices.AbstractXBeeDevice method), 238
get_protocol() (digi.xbee.devices.CellularDevice method), 361
get_protocol() (digi.xbee.devices.DigiMeshDevice method), 284
get_protocol() (digi.xbee.devices.DigiPointDevice method), 303
get_protocol() (digi.xbee.devices.IPDevice method), 353
get_protocol() (digi.xbee.devices.LPWANDevice method), 388
get_protocol() (digi.xbee.devices.NBLoTDevice method), 398
get_protocol() (digi.xbee.devices.Raw802Device method), 266
get_protocol() (digi.xbee.devices.RemoteDigiMeshDevice method), 464
get_protocol() (digi.xbee.devices.RemoteDigiPointDevice method), 476
get_protocol() (digi.xbee.devices.RemoteRaw802Device method), 452
get_protocol() (digi.xbee.devices.RemoteXBeeDevice method), 446
get_protocol() (digi.xbee.devices.RemoteZigBeeDevice method), 488
get_protocol() (digi.xbee.devices.WiFiDevice method), 416
get_protocol() (digi.xbee.devices.XBeeDevice method), 259
get_protocol() (digi.xbee.devices.ZigBeeDevice method), 322

Index 585
get_pwm_duty_cycle()
(digi.xbee.devices.AbstractXBeeDevice method), 243
get_pwm_duty_cycle()
(digi.xbee.devices.CellularDevice method), 370
get_pwm_duty_cycle()
(digi.xbee.devices.DigiMeshDevice method), 292
get_pwm_duty_cycle()
(digi.xbee.devices.DigiPointDevice method), 311
get_pwm_duty_cycle()
(digi.xbee.devices.IPDevice method), 353
get_pwm_duty_cycle()
(digi.xbee.devices.LPWANDevice method), 388
get_pwm_duty_cycle()
(digi.xbee.devices.NBIoTDevice method), 405
get_pwm_duty_cycle()
(digi.xbee.devices.Raw802Device method), 273
get_pwm_duty_cycle()
(digi.xbee.devices.RemoteDigiMeshDevice method), 469
get_pwm_duty_cycle()
(digi.xbee.devices.RemoteDigiPointDevice method), 481
get_pwm_duty_cycle()
(digi.xbee.devices.RemoteRaw802Device method), 457
get_pwm_duty_cycle()
(digi.xbee.devices.RemoteXBeeDevice method), 446
get_pwm_duty_cycle()
(digi.xbee.devices.RemoteZigBeeDevice method), 493
get_pwm_duty_cycle()
(digi.xbee.devices.WiFiDevice method), 429
get_pwm_duty_cycle()
(digi.xbee.devices.XBeeDevice method), 259
get_pwm_duty_cycle()
(digi.xbee.devices.ZigBeeDevice method), 332
get_queue()
(digi.xbee.reader.PacketListener method), 545
get_read_timeout()
(digi.xbee.serial.XBeeSerialPort method), 552
get_serial_port()
(digi.xbee.devices.RemoteDigiPointDevice method), 482
get_serial_port()
(digi.xbee.devices.RemoteRaw802Device method), 458
get_serial_port()
(digi.xbee.devices.RemoteXBeeDevice method), 441
get_serial_port()
(digi.xbee.devices.RemoteZigBeeDevice method), 494
get_sync_ops_timeout()
(digi.xbee.devices.AbstractXBeeDevice method), 240
get_sync_ops_timeout()
(digi.xbee.devices.CellularDevice method), 370
get_sync_ops_timeout()
(digi.xbee.devices.DigiMeshDevice method), 292
get_sync_ops_timeout()
(digi.xbee.devices.DigiPointDevice method), 311
get_sync_ops_timeout()
(digi.xbee.devices.IPDevice method), 354
get_sync_ops_timeout()
(digi.xbee.devices.LPWANDevice method), 388
get_sync_ops_timeout()
(digi.xbee.devices.NBIoTDevice method), 405
get_sync_ops_timeout()
(digi.xbee.devices.Raw802Device method), 273
get_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiMeshDevice method), 470
get_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiPointDevice method), 482
get_sync_ops_timeout()
(digi.xbee.devices.RemoteRaw802Device method), 458
get_sync_ops_timeout()
(digi.xbee.devices.RemoteXBeeDevice method), 447
get_sync_ops_timeout()
(digi.xbee.devices.RemoteZigBeeDevice method), 494
get_sync_ops_timeout()
(digi.xbee.devices.WiFiDevice method), 429
get_sync_ops_timeout()
(digi.xbee.devices.XBeeDevice method), 470
has_sync_ops_timeout()
(digi.xbee.devices.ZigBeeDevice method), 332
get_wifi_ai_status()
(digi.xbee.devices.WiFiDevice method), 416
get_xbee_device_callbacks()
(digi.xbee.devices.CellularDevice method), 293
get_xbee_device_callbacks()
(digi.xbee.devices.DigiMeshDevice method), 293
get_xbee_device_callbacks()
(digi.xbee.devices.DigiPointDevice method), 312
get_xbee_device_callbacks()
(digi.xbee.devices.IPDevice method), 354
get_xbee_device_callbacks()
(digi.xbee.devices.LPWANDevice method), 389
get_xbee_device_callbacks()
(digi.xbee.devices.NBIoTDevice method), 430
get_xbee_device_callbacks()
(digi.xbee.devices.Raw802Device method), 274
get_xbee_device_callbacks()
(digi.xbee.devices.XBeeDevice method), 252
get_xbee_device_callbacks()
(digi.xbee.devices.ZigBeeDevice method), 332

HardwareVersion (class in digi.xbee.models.hw), 97
has_analog_value()
(digi.xbee.io.IOSample method), 531
has_analog_values()
(digi.xbee.io.IOSample method), 531
has_devices()
(digi.xbee.devices.DigiMeshNetwork method), 519
has_devices()
(digi.xbee.devices.DigiPointNetwork method), 525
has_devices()
(digi.xbee.devices.Raw802Network method), 514
has_devices()
(digi.xbee.devices.XBeeNetwork method), 501
has_devices()
(digi.xbee.devices.ZigBeeNetwork method), 509
has_digital_value()
(digi.xbee.io.IOSample method), 531
has_digital_values()
(digi.xbee.io.IOSample method), 531
has_explicit_packets()
(digi.xbee.devices.CellularDevice method), 371
has_explicit_packets()
(digi.xbee.devices.DigiMeshDevice method), 293
has_explicit_packets()
(digi.xbee.devices.DigiPointDevice method), 312
has_explicit_packets()
(digi.xbee.devices.IPDevice method), 354
has_explicit_packets()
(digi.xbee.devices.LPWANDevice method), 389
has_explicit_packets()
(digi.xbee.devices.NBIoTDevice method), 406
has_explicit_packets()
(digi.xbee.devices.Raw802Device method), 274
has_explicit_packets()
(digi.xbee.devices.WiFiDevice method), 430
has_explicit_packets()
(digi.xbee.devices.XBeeDevice method), 252
has_explicit_packets()
(digi.xbee.devices.ZigBeeDevice method), 332
has_packets()
(digi.xbee.devices.CellularDevice method), 371
has_packets()
(digi.xbee.devices.DigiMeshDevice method), 293
has_packets()
(digi.xbee.devices.DigiPointDevice method), 293
has_packets()
(digi.xbee.devices.IPDevice method), 312
has_packets()
(digi.xbee.devices.LPWANDevice method), 354
has_packets()
(digi.xbee.devices.NBIoTDevice method), 406
has_packets()
(digi.xbee.devices.Raw802Device method), 274
has_packets()
(digi.xbee.devices.WiFiDevice method), 430
has_packets()
(digi.xbee.devices.XBeeDevice method), 252
has_packets()
(digi.xbee.devices.ZigBeeDevice method), 332
has_power_supply_value()
(digi.xbee.io.IOSample method), 531
has_pwm_capability()
(digi.xbee.io.IOLine method), 529
hex_string_to_bytes() (in module digi.xbee.util.utils), 232
hex_to_string() (in module digi.xbee.util.utils), 234

ident (digi.xbee.reader.PacketListener attribute), 548
index (digi.xbee.io.iOLine attribute), 529
index() (digi.xbee.reader.BluetoothDataReceived method), 543
index() (digi.xbee.reader.DataReceived method), 535
index() (digi.xbee.reader.DeviceDiscovered method), 538
index() (digi.xbee.reader.DiscoveryProcessFinished method), 538
index() (digi.xbee.reader.ExplicitDataReceived method), 539
index() (digi.xbee.reader.IOSampleReceived method), 537
index() (digi.xbee.reader.IPDataReceived method), 540
index() (digi.xbee.reader.MicroPythonDataReceived method), 544
index() (digi.xbee.reader.ModemStatusReceived method), 536
index() (digi.xbee.reader.PacketReceived method), 534
index() (digi.xbee.reader.RelayDataReceived method), 542
index() (digi.xbee.reader.SMSSent method), 541
index() (digi.xbee.reader.XBeEvent method), 533
insert() (digi.xbee.reader.BluetoothDataReceived method), 543
insert() (digi.xbee.reader.DataReceived method), 535
insert() (digi.xbee.reader.DeviceDiscovered method), 538
insert() (digi.xbee.reader.DiscoveryProcessFinished method), 539
insert() (digi.xbee.reader.ExplicitDataReceived method), 539
insert() (digi.xbee.reader.IOSampleReceived method), 537
insert() (digi.xbee.reader.IPDataReceived method), 540
insert() (digi.xbee.reader.MicroPythonDataReceived method), 544
insert() (digi.xbee.reader.ModemStatusReceived method), 536
insert() (digi.xbee.reader.PacketReceived method), 534
insert() (digi.xbee.reader.RelayDataReceived method), 542
insert() (digi.xbee.reader.SMSSent method), 541
digi.xbee.reader.XBeEvent method), 533
insert() (digi.xbee.reader.BluetoothDataReceived method), 543
insert() (digi.xbee.reader.DataReceived method), 535
insert() (digi.xbee.reader.DeviceDiscovered method), 538
insert() (digi.xbee.reader.DiscoveryProcessFinished method), 539
insert() (digi.xbee.reader.ExplicitDataReceived method), 539
insert() (digi.xbee.reader.ISSampleReceived method), 537
insert() (digi.xbee.reader.IPDataReceived method), 540
insert() (digi.xbee.reader.MicroPythonDataReceived method), 544
insert() (digi.xbee.reader.ModemStatusReceived method), 536
insert() (digi.xbee.reader.PacketReceived method), 534
insert() (digi.xbee.reader.RelayDataReceived method), 542
insert() (digi.xbee.reader.SMSSent method), 541
insert() (digi.xbee.reader.XBeEvent method), 533
int_to_ascii() (in module digi.xbee.util.utils), 233
int_to_bytess() (in module digi.xbee.util.utils), 232
int_to_length() (in module digi.xbee.util.utils), 234
InvalidConfigurationException, 527
InvalidOperatingModeException, 527
InvalidPacketException, 527
io_sample (digi.xbee.packets.common.IODataSampleRxIndicatorPacket attribute), 165
io_sample (digi.xbee.packets.raw.RX16IOPacket attribute), 216
io_sample (digi.xbee.packets.raw.RX64IOPacket attribute), 212
io_sample (digi.xbee.packets.wifi.IODataSampleRxIndicatorWiFiPacket attribute), 223
IOPacketRxIndicatorPacket (class in digi.xbee.packets.common), 163
IOPacketRxIndicatorWiFiPacket (class in digi.xbee.packets.wifi), 222
IOLine (class in digi.xbee.io), 528
IOMode (class in digi.xbee.io), 532
IOMode (class in digi.xbee.io), 529
IOSampleReceived (class in digi.xbee.reader), 536
IOValue (class in digi.xbee.io), 529
ip_addr (digi.xbee.models.message.IPMessage attribute), 106
ip_protocol (digi.xbee.packets.network.RXIPv4Packet attribute), 192
ip_protocol (digi.xbee.packets.network.TXIPv4Packet attribute), 196
IPAddressMode (class in digi.xbee.models.mode), 100
IPDataReceived (class in digi.xbee.reader), 540
IPDevice (class in digi.xbee.devices), 342
IPMessage (class in digi.xbee.models.message), 105
IPProtocol (class in digi.xbee.models.protocol), 112
is_alive() (digi.xbee.reader.PacketListener method), 548
is_apply_changes_enabled() (digi.xbee.devices.AbstractXBeeDevice method), 239
is_apply_changes_enabled() (digi.xbee.devices.CellularDevice method), 371
is_apply_changes_enabled() (digi.xbee.devices.DigiMeshDevice method), 293
is_apply_changes_enabled() (digi.xbee.devices.DigiPointDevice method), 312
is_apply_changes_enabled() (digi.xbee.devices.IPDevice method), 354
is_apply_changes_enabled() (digi.xbee.devices.WiFiDevice method), 354
is_apply_changes_enabled()
(digi.xbee.devices.LPWANDevice method), 170
is_apply_changes_enabled()
(digi.xbee.devices.NBIoTDevice method), 172
is_apply_changes_enabled()
(digi.xbee.devices.Raw802Device method), 164
is_apply_changes_enabled()
(digi.xbee.devices.RemoteDigiMeshDevice method), 147
is_apply_changes_enabled()
(digi.xbee.devices.RemoteDigiPointDevice method), 150
is_apply_changes_enabled()
(digi.xbee.devices.RemoteRaw802Device method), 154
is_apply_changes_enabled()
(digi.xbee.devices.RemoteXBeeDevice method), 157
is_apply_changes_enabled()
(digi.xbee.devices.ZigBeeDevice method), 160
is_bit_enabled()
(in module digi.xbee.util.utils), 157
is_broadcast()
(digi.xbee.packets.common.ExplicitRXIndicatorPacket method), 159
is_broadcast()
(digi.xbee.packets.common.IODataSampleRxIndicatorPacket method), 161
is_broadcast()
(digi.xbee.packets.common.ModemStatusPacket method), 163
is_broadcast()
(digi.xbee.packets.common.ReceivePacket method), 166
is_broadcast()
(digi.xbee.packets.common.RemoteATCommandPacket method), 169
is_broadcast()
(digi.xbee.packets.common.RemoteATCommandResponsePacket method), 172
is_broadcast()
(digi.xbee.packets.common.TransmitPacket method), 175
is_broadcast()
(digi.xbee.packets.common.TransmitStatusPacket method), 178
is_broadcast()
(digi.xbee.packets.devicecloud.DeviceRequestPacket method), 181
is_broadcast()
(digi.xbee.packets.devicecloud.DeviceResponsePacket method), 184
is_broadcast()
(digi.xbee.packets.devicecloud.DeviceResponseStatusPacket method), 187
is_broadcast()
(digi.xbee.packets.devicecloud.FrameErrorPacket method), 190
is_broadcast()
(digi.xbee.packets.devicecloud.SendDataRequestPacket method), 193
is_broadcast()
(digi.xbee.packets.devicecloud.SendDataResponsePacket method), 196
is_broadcast()
(digi.xbee.packets.devicecloud.SendDataStatusPacket method), 199
is_broadcast()
(digi.xbee.packets.devicecloud.SendDataStatusPacket method), 202
is_broadcast()
(digi.xbee.packets.relay.UserDataRelayOutputPacket method), 205
is_broadcast()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 208
is_broadcast()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 211
is_broadcast()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 214
is_broadcast()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 217
is_broadcast()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 220
is_broadcast()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket method), 223
log (digi.xbee.devices.RemoteZigBeeDevice
attribute), 494
log (digi.xbee.devices.ZigBeeDevice
attribute), 333
LOG_PATTERN (digi.xbee.devices.AbstractXBeeDevice
attribute), 235
LPWANDevice (class in digi.xbee.devices), 380

M
MicroPythonDataReceived (class in
digi.xbee.reader), 543
min_io_sample_payload()
(digi.xbee.io.IOSample static method), 531
modem_status
digi.xbee.packets.common.ModemStatusPacket
attribute), 162
ModemStatus
(digi.xbee.models.status), 115
ModemStatusPacket
(digi.xbee.packets.common), 160
ModemStatusReceived
(digi.xbee.reader), 535

N
digi.xbee.reader.PacketListener attribute), 548
NBIoTDevice
(digi.xbee.devices), 397
ND_PACKET_FINISH
(digi.xbee.devices.XBeeNetwork
attribute), 500
ND_PACKET_REMOTE
(digi.xbee.devices.XBeeNetwork
attribute), 500
needs_id()
(digi.xbee.packets.base.GenericXBeePacket
method), 126
needs_id()
(digi.xbee.packets.base.UnknownXBeePacket
method), 130
needs_id()
(digi.xbee.packets.base.XBeeAPIPacket
method), 124
needs_id()
(digi.xbee.packets.cellular.RXSMSPacket
method), 131
needs_id()
(digi.xbee.packets.cellular.TXSMSPacket
method), 134
needs_id()
(digi.xbee.packets.common.ATCommPacket
method), 137
needs_id()
(digi.xbee.packets.common.ATCommQueuePacket
method), 140
needs_id()
(digi.xbee.packets.common.ATCommResponsePacket
method), 143
needs_id()
(digi.xbee.packets.common.ExplicitAddressingPacket
method), 168
needs_id()
(digi.xbee.packets.common.ExplicitRXIndicatorPacket
method), 173
needs_id()
(digi.xbee.packets.common.IODataSampleRxIndicatorPacket
method), 164
needs_id()
(digi.xbee.packets.common.ModemStatusPacket
method), 161

needs_id()
(digi.xbee.packets.common.ReceivePacket
method), 146
needs_id()
(digi.xbee.packets.common.RemoteATCommandPacket
method), 149
needs_id()
(digi.xbee.packets.common.RemoteATCommandResponsePacket
method), 152
needs_id()
(digi.xbee.packets.common.TransmitPacket
method), 156
needs_id()
(digi.xbee.packets.common.TransmitStatusPacket
method), 159
needs_id()
(digi.xbee.packets.devicecloud.DeviceRequestPacket
method), 175
needs_id()
(digi.xbee.packets.devicecloud.DeviceResponsePacket
method), 178
needs_id()
(digi.xbee.packets.devicecloud.DeviceResponseStatusPacket
method), 180
needs_id()
(digi.xbee.packets.devicecloud.SendDataRequestPacket
method), 186
needs_id()
(digi.xbee.packets.devicecloud.SendDataResponsePacket
method), 190
needs_id()
(digi.xbee.packets.network.RXIPv4Packet
method), 191
needs_id()
(digi.xbee.packets.network.TXIPv4Packet
method), 194
needs_id()
(digi.xbee.packets.raw.RX16IOPacket
method), 216
needs_id()
(digi.xbee.packets.raw.RX16Packet
method), 208
needs_id()
(digi.xbee.packets.raw.RX64IOPacket
method), 211
needs_id()
(digi.xbee.packets.raw.RX64Packet
method), 206
needs_id()
(digi.xbee.packets.tx16Packet
method), 200
needs_id()
(digi.xbee.packets.tx64Packet
method), 197
needs_id()
(digi.xbee.packets.raw.TXStatusPacket
method), 203
needs_id()
(digi.xbee.packets.reader/UserDataRelayOutputPacket
method), 220
needs_id()
(digi.xbee.packets.reader/UserDataRelayPacket
method), 217
needs_id()
(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket
method), 223
needs_id()
(digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket
method), 229
needs_id()
(digi.xbee.packets.wifi.RemoteATCommandWifiPacket
method), 226
NetworkDiscoveryStatus
(digi.xbee.models.status), 120
(digi.xbee.models.options.ReceiveOptions
attribute), 107
NONE (digixbee.models.options.RemoteATCmdOptions attribute), 110
NONE (digixbee.models.options.TransmitOptions attribute), 108

open() (digixbee.devices.CellularDevice method), 361
open() (digixbee.devices.DigiMeshDevice method), 284
open() (digixbee.devices.DigiPointDevice method), 303
open() (digixbee.devices.IPDevice method), 355
open() (digixbee.devices.LPWANDevice method), 390
open() (digixbee.devices.NBIoTDevice method), 398
open() (digixbee.devices.Raw802Device method), 265
open() (digixbee.devices.WiFiDevice method), 415
open() (digixbee.devices.XBeeDevice method), 248
open() (digixbee.devices.ZigBeeDevice method), 322
operating_mode (digixbee.devices.CellularDevice attribute), 372
operating_mode (digixbee.devices.DigiMeshDevice attribute), 294
operating_mode (digixbee.devices.DigiPointDevice attribute), 313
operating_mode (digixbee.devices.IPDevice attribute), 355
operating_mode (digixbee.devices.LPWANDevice attribute), 390
operating_mode (digixbee.devices.NBIoTDevice attribute), 407
operating_mode (digixbee.devices.Raw802Device attribute), 275
operating_mode (digixbee.devices.WiFiDevice attribute), 430
operating_mode (digixbee.devices.XBeeDevice attribute), 254
operating_mode (digixbee.devices.ZigBeeDevice attribute), 333
OperatingMode (class in digixbee.models.mode), 99
OperationNotSupportedException, 144
options (digixbee.packets.devicecloud.SendDataRequestPacket attribute), 186
OPTIONS_CLOSE_SOCKET (digixbee.packets.network.TXIPv4Packet attribute), 194
OPTIONS_LEAVE_SOCKET_OPEN (digixbee.packets.network.TXIPv4Packet attribute), 194
output() (digixbee.packets.base.GenericXBeePacket method), 128
output() (digixbee.packets.base.UnknownXBeePacket method), 130
output() (digixbee.packets.base.XBeeAPIPacket method), 125
output() (digixbee.packets.base.XBeePacket method), 122
output() (digixbee.packets.cellular.RXSMSPacket method), 133
output() (digixbee.packets.cellular.TXSMSPacket method), 136
output() (digixbee.packets.common.ATCommPacket method), 138
output() (digixbee.packets.common.ATCommQueuePacket method), 141
output() (digixbee.packets.common.ATCommResponsePacket method), 144
output() (digixbee.packets.common.ExplicitAddressingPacket method), 170
output() (digixbee.packets.common.ExplicitRXIndicatorPacket method), 172
output() (digixbee.packets.common.IODataSampleRXIndicatorPacket method), 166
output() (digixbee.packets.common.ModemStatusPacket method), 163
output() (digixbee.packets.common.ReceivePacket method), 147
output() (digixbee.packets.common.RemoteATCommandPacket method), 150
output() (digixbee.packets.common.RemoteATCommandResponsePacket method), 154
output() (digixbee.packets.common.TransmitPacket method), 157
output() (digixbee.packets.common.TransmitStatusPacket method), 160
output() (digixbee.packets.devicecloud.DeviceRequestPacket method), 176
output() (digixbee.packets.devicecloud.DeviceResponsePacket method), 179
output() (digixbee.packets.devicecloud.DeviceResponseStatusPacket method), 182
output() (digixbee.packets.devicecloud.FrameErrorPacket method), 184
output() (digixbee.packets.devicecloud.SendDataRequestPacket method), 187
output() (digixbee.packets.devicecloud.SendDataResponsePacket method), 189
output() (digixbee.packets.network.RXIPv4Packet method), 193
output() (digixbee.packets.network.TXIPv4Packet method), 196
output() (digixbee.packets.raw.RX161OPacket method), 215
output() (digixbee.packets.raw.RX16Packet method), 210
output() (digixbee.packets.raw.RX641OPacket method), 213
output() (digixbee.packets.raw.RX64Packet method), 207
Index 593

PACKET_ACKNOWLEDGED
(digi.xbee.models.options.ReceiveOptions attribute), 107
PacketListener (class in digi.xbee.reader), 544
PacketReceived (class in digi.xbee.reader), 533
parameter (digi.xbee.models.atcomm.ATCommand attribute), 97
parameter (digi.xbee.packets.common.ATCommPacket attribute), 137
parameter (digi.xbee.packets.common.ATCommQueuePacket attribute), 140
parameter (digi.xbee.packets.common.RemoteATCommandPacket attribute), 149
parameter (digi.xbee.packets.wifi.RemoteATCommandWiFiPacket attribute), 226
path (digi.xbee.packets.devicecloud.SendDataRequestPacket attribute), 186
PATTERN_PHONE_NUMBER (in module digi.xbee.packets.cellular), 131
phone_number (digi.xbee.models.message.SMSMessage attribute), 106
phone_number (digi.xbee.packets.cellular.RXSMSPacket attribute), 132
phone_number (digi.xbee.packets.cellular.TXSMSPacket attribute), 136
POINT_MULTIPOINT_MODE (digi.xbee.models.options.TransmitOptions attribute), 109
pop () (digi.xbee.reader.BluetoothDataReceived method), 543
pop () (digi.xbee.reader.DataReceived method), 535
pop () (digi.xbee.reader.DeviceDiscovered method), 538
pop () (digi.xbee.reader.DiscoveryProcessFinished method), 539
pop () (digi.xbee.reader.ExplicitDataReceived method), 539
pop () (digi.xbee.reader.IO SAMPLE Received method), 537
pop () (digi.xbee.reader.IPDataReceived method), 540
pop () (digi.xbee.reader.MicroPythonDataReceived method), 544
pop () (digi.xbee.reader.ModemStatusReceived method), 536
pop () (digi.xbee.reader.PacketReceived method), 534
pop () (digi.xbee.reader.RelayDataReceived method), 542
pop () (digi.xbee.reader.RelayDataReceived method), 533
put ( ) (digi.xbee.reader.XBeeQueue method), 550
put_nowait () (digi.xbee.reader.XBeeQueue method), 551
PWM (digi.xbee.io.IOMode attribute), 532
read_at_command (digi.xbee.io.IOLine attribute), 529
read_data() (digi.xbee.reader.XBeeQueue method), 551
read_data() (digi.xbee.devices.CellularDevice method), 372
read_data() (digi.xbee.devices.DigiMeshDevice method), 294
read_data() (digi.xbee.devices.DigiPointDevice method), 313
read_data() (digi.xbee.devices.IPDevice method), 347
read_data() (digi.xbee.devices.LPWANDevice method), 390
read_data() (digi.xbee.devices.NBIoTDevice method), 407
read_data() (digi.xbee.devices.Raw802Device method), 275

XBee Python Library Documentation, Release 1.2.0

P
 PacketListener (class in digi.xbee.reader), 544
 PacketReceived (class in digi.xbee.reader), 533
 parameter (digi.xbee.models.atcomm.ATCommand attribute), 97
 parameter (digi.xbee.packets.common.ATCommPacket attribute), 137
 parameter (digi.xbee.packets.common.ATCommQueuePacket attribute), 140
 parameter (digi.xbee.packets.common.RemoteATCommandPacket attribute), 149
 parameter (digi.xbee.packets.wifi.RemoteATCommandWiFiPacket attribute), 226
 path (digi.xbee.packets.devicecloud.SendDataRequestPacket attribute), 186
 PATTERN_PHONE_NUMBER (in module digi.xbee.packets.cellular), 131
 phone_number (digi.xbee.models.message.SMSMessage attribute), 106
 phone_number (digi.xbee.packets.cellular.RXSMSPacket attribute), 132
 phone_number (digi.xbee.packets.cellular.TXSMSPacket attribute), 136
 POINT_MULTIPOINT_MODE (digi.xbee.models.options.TransmitOptions attribute), 109
 pop () (digi.xbee.reader.BluetoothDataReceived method), 543
 pop () (digi.xbee.reader.DataReceived method), 535
 pop () (digi.xbee.reader.DeviceDiscovered method), 538
 pop () (digi.xbee.reader.DiscoveryProcessFinished method), 539
 pop () (digi.xbee.reader.ExplicitDataReceived method), 539
 pop () (digi.xbee.reader.IO SAMPLE Received method), 537
 pop () (digi.xbee.reader.IPDataReceived method), 540
 pop () (digi.xbee.reader.MicroPythonDataReceived method), 544
 pop () (digi.xbee.reader.ModemStatusReceived method), 536
 pop () (digi.xbee.reader.PacketReceived method), 534
 pop () (digi.xbee.reader.RelayDataReceived method), 542
 pop () (digi.xbee.reader.RelayDataReceived method), 533
 put ( ) (digi.xbee.reader.XBeeQueue method), 550
 put_nowait () (digi.xbee.reader.XBeeQueue method), 551
 PWM (digi.xbee.io.IOMode attribute), 532
 read_at_command (digi.xbee.io.IOLine attribute), 529
 read_data() (digi.xbee.reader.XBeeQueue method), 551
 read_data() (digi.xbee.devices.CellularDevice method), 372
 read_data() (digi.xbee.devices.DigiMeshDevice method), 294
 read_data() (digi.xbee.devices.DigiPointDevice method), 313
 read_data() (digi.xbee.devices.IPDevice method), 347
 read_data() (digi.xbee.devices.LPWANDevice method), 390
 read_data() (digi.xbee.devices.NBIoTDevice method), 407
 read_data() (digi.xbee.devices.Raw802Device method), 275

XBee Python Library Documentation, Release 1.2.0

P
 PacketListener (class in digi.xbee.reader), 544
 PacketReceived (class in digi.xbee.reader), 533
 parameter (digi.xbee.models.atcomm.ATCommand attribute), 97
 parameter (digi.xbee.packets.common.ATCommPacket attribute), 137
 parameter (digi.xbee.packets.common.ATCommQueuePacket attribute), 140
 parameter (digi.xbee.packets.common.RemoteATCommandPacket attribute), 149
 parameter (digi.xbee.packets.wifi.RemoteATCommandWiFiPacket attribute), 226
 path (digi.xbee.packets.devicecloud.SendDataRequestPacket attribute), 186
 PATTERN_PHONE_NUMBER (in module digi.xbee.packets.cellular), 131
 phone_number (digi.xbee.models.message.SMSMessage attribute), 106
 phone_number (digi.xbee.packets.cellular.RXSMSPacket attribute), 132
 phone_number (digi.xbee.packets.cellular.TXSMSPacket attribute), 136
 POINT_MULTIPOINT_MODE (digi.xbee.models.options.TransmitOptions attribute), 109
 pop () (digi.xbee.reader.BluetoothDataReceived method), 543
 pop () (digi.xbee.reader.DataReceived method), 535
 pop () (digi.xbee.reader.DeviceDiscovered method), 538
 pop () (digi.xbee.reader.DiscoveryProcessFinished method), 539
 pop () (digi.xbee.reader.ExplicitDataReceived method), 539
 pop () (digi.xbee.reader.IO SAMPLE Received method), 537
 pop () (digi.xbee.reader.IPDataReceived method), 540
 pop () (digi.xbee.reader.MicroPythonDataReceived method), 544
 pop () (digi.xbee.reader.ModemStatusReceived method), 536
 pop () (digi.xbee.reader.PacketReceived method), 534
 pop () (digi.xbee.reader.RelayDataReceived method), 542
 pop () (digi.xbee.reader.RelayDataReceived method), 533
 put ( ) (digi.xbee.reader.XBeeQueue method), 550
 put_nowait () (digi.xbee.reader.XBeeQueue method), 551
 PWM (digi.xbee.io.IOMode attribute), 532
 read_at_command (digi.xbee.io.IOLine attribute), 529
 read_data() (digi.xbee.reader.XBeeQueue method), 551
 read_data() (digi.xbee.devices.CellularDevice method), 372
 read_data() (digi.xbee.devices.DigiMeshDevice method), 294
 read_data() (digi.xbee.devices.DigiPointDevice method), 313
 read_data() (digi.xbee.devices.IPDevice method), 347
 read_data() (digi.xbee.devices.LPWANDevice method), 390
 read_data() (digi.xbee.devices.NBIoTDevice method), 407
 read_data() (digi.xbee.devices.Raw802Device method), 275
read_data() (digi.xbee.devices.WiFiDevice method), 431
read_data() (digi.xbee.devices.XBeeDevice method), 251
read_data() (digi.xbee.devices.ZigBeeDevice method), 333
read_data_from() (digi.xbee.devices.CellularDevice method), 372
read_data_from() (digi.xbee.devices.DigiMeshDevice method), 294
read_data_from() (digi.xbee.devices.DigiPointDevice method), 313
read_data_from() (digi.xbee.devices.IPDevice method), 347
read_data_from() (digi.xbee.devices.LPWANDevice method), 390
read_data_from() (digi.xbee.devices.NBLoTDevice method), 407
read_data_from() (digi.xbee.devices.Raw802Device method), 275
read_data_from() (digi.xbee.devices.XBeeDevice method), 252
read_data_from() (digi.xbee.devices.ZigBeeDevice method), 334
read_device_info() (digi.xbee.devices.AbstractXBeeDevice method), 237
read_device_info() (digi.xbee.devices.CellularDevice method), 362
read_device_info() (digi.xbee.devices.DigiMeshDevice method), 295
read_device_info() (digi.xbee.devices.DigiPointDevice method), 314
read_device_info() (digi.xbee.devices.IPDevice method), 343
read_device_info() (digi.xbee.devices.LPWANDevice method), 390
read_device_info() (digi.xbee.devices.NBLoTDevice method), 407
read_device_info() (digi.xbee.devices.Raw802Device method), 276
read_device_info() (digi.xbee.devices.RemoteDigiMeshDevice method), 470
read_device_info() (digi.xbee.devices.RemoteDigiPointDevice method), 447
read_device_info() (digi.xbee.devices.WiFiDevice method), 431
read_device_info() (digi.xbee.devices.XBeeDevice method), 260
read_device_info() (digi.xbee.devices.ZigBeeDevice method), 334
read_existing() (digi.xbee.serial.XBeeSerialPort method), 552
read_expl_data() (digi.xbee.devices.DigiMeshDevice method), 285
read_expl_data() (digi.xbee.devices.DigiPointDevice method), 304
read_expl_data() (digi.xbee.devices.ZigBeeDevice method), 323
read_expl_data_from() (digi.xbee.devices.DigiMeshDevice method), 285
read_expl_data_from() (digi.xbee.devices.DigiPointDevice method), 304
read_expl_data_from() (digi.xbee.devices.ZigBeeDevice method), 323
read_io_sample() (digi.xbee.devices.AbstractXBeeDevice method), 242
read_io_sample() (digi.xbee.devices.CellularDevice method), 372
read_io_sample() (digi.xbee.devices.DigiMeshDevice method), 295
read_io_sample() (digi.xbee.devices.DigiPointDevice method), 314
read_io_sample() (digi.xbee.devices.IPDevice method), 355
read_io_sample() (digi.xbee.devices.LPWANDevice method), 390
read_io_sample() (digi.xbee.devices.NBLoTDevice method), 407
read_io_sample() (digi.xbee.devices.Raw802Device method), 276
read_io_sample() (digi.xbee.devices.RemoteDigiMeshDevice method), 471
read_io_sample() (digi.xbee.devices.RemoteDigiPointDevice method), 471
remove_device() (digi.xbee.devices.Raw802Network method), 514
remove_device() (digi.xbee.devices.XBeeNetwork method), 505
remove_device() (digi.xbee.devices.ZigBeeNetwork method), 509

REPEATER_MODE (digi.xbee.models.options.TransmitOptions attribute), 109
request_data (digi.xbee.packets.devicecloud.DeviceRequestPacket method), 536
request_data (digi.xbee.packets.devicecloud.DeviceResponsePacket method), 534
request_id (digi.xbee.packets.devicecloud.DeviceRequestPacket method), 542
request_id (digi.xbee.packets.devicecloud.DeviceResponsePacket method), 541
reset() (digi.xbee.devices.AbstractXBeeDevice method), 237
reset() (digi.xbee.devices.CellularDevice method), 373
reset() (digi.xbee.devices.DigiMeshDevice method), 295
reset() (digi.xbee.devices.DigiPointDevice method), 314
reset() (digi.xbee.devices.IPDevice method), 355
reset() (digi.xbee.devices.LPWANDevice method), 391
reset() (digi.xbee.devices.NBIoTDevice method), 408
reset() (digi.xbee.devices.Raw802Device method), 276
reset() (digi.xbee.devices.RemoteDigiMeshDevice method), 471
reset() (digi.xbee.devices.RemoteDigiPointDevice method), 483
reset() (digi.xbee.devices.RemoteRaw802Device method), 459
reset() (digi.xbee.devices.RemoteXBeeDevice method), 441
reset() (digi.xbee.devices.RemoteZigBeeDevice method), 495
reset() (digi.xbee.devices.WiFiDevice method), 432
reset() (digi.xbee.devices.XBeeDevice method), 253
response (digi.xbee.models.atcomm.ATCommandResponse attribute), 97
reverse() (digi.xbee.reader.bluetooth.DeviceDataReceived method), 543
reverse() (digi.xbee.reader.DataReceived method), 535
reverse() (digi.xbee.reader.DeviceDiscovered method), 538
reverse() (digi.xbee.reader.DeviceDiscovered method), 538
reverse() (digi.xbee.reader.DiscoveryProcessFinished method), 539
reverse() (digi.xbee.reader.ExplicitDataReceived method), 540
reverse() (digi.xbee.reader.IOReceived command method), 537
reverse() (digi.xbee.reader.IPDataReceived method), 540
reverse() (digi.xbee.reader.ModemStatusReceived method), 544
reverse() (digi.xbee.reader.RelayDataReceived method), 546
reverse() (digi.xbee.reader.RelayDataReceived method), 546
reverse() (digi.xbee.reader.SMSReceived method), 223
reverse() (digi.xbee.reader.WiFiEvent method), 223
rf_data (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 170
rf_data (digi.xbee.packets.common.ExplicitRXIndicatorPacket attribute), 173
rf_data (digi.xbee.packets.common.IODataSampleRxIndicatorPacket attribute), 165
rf_data (digi.xbee.packets.common.ReceivePacket attribute), 146
rf_data (digi.xbee.packets.common.TransmitPacket attribute), 156
rf_data (digi.xbee.packets.raw.RX161OPacket attribute), 216
rf_data (digi.xbee.packets.raw.RX16Packet attribute), 209
rf_data (digi.xbee.packets.raw.RX64IOPacket attribute), 212
rf_data (digi.xbee.packets.raw.RX64Packet attribute), 206
rf_data (digi.xbee.packets.raw.TX16Packet attribute), 200
rf_data (digi.xbee.packets.raw.TX64Packet attribute), 198
rf_data (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket attribute), 223
rssi (digi.xbee.packets.raw.RX161OPacket attribute), 216
rssi (digi.xbee.packets.raw.RX16Packet attribute), 209
rssi (digi.xbee.packets.raw.RX641OPacket attribute), 211
rssi (digi.xbee.packets.raw.RX64Packet attribute), 206
rssi (digi.xbee.packets.wifi.IODataSampleRxIndicatorWifiPacket attribute), 223
run() (digi.xbee.reader.PacketListener method), 545
RX161OPacket (class in digi.xbee.packets.raw), 213
RX16Packet (class in digi.xbee.packets.raw), 207
RX641OPacket (class in digi.xbee.packets.raw), 210
RX64Packet (class in digi.xbee.packets.raw), 204
RXIPv4Packet (class in digi.xbee.packets.network), 223
RXSMSPacket (class in digi.xbee.packets.cellular), 131

S

scan_access_points()
(digi.xbee.devices.WiFiDevice method), 417

send_bluetooth_data()
(digi.xbee.devices.CellularDevice method), 373

send_bluetooth_data()
(digi.xbee.devices.DigiMeshDevice method), 296

send_bluetooth_data()
(digi.xbee.devices.DigiPointDevice method), 315

send_bluetooth_data()
(digi.xbee.devices.IPDevice method), 355

send_bluetooth_data()
(digi.xbee.devices.LPWANDevice method), 391

send_bluetooth_data()
(digi.xbee.devices.NBIoTDevice method), 408

send_bluetooth_data()
(digi.xbee.devices.Raw802Device method), 277

send_bluetooth_data()
(digi.xbee.devices.WiFiDevice method), 432

send_bluetooth_data()
(digi.xbee.devices.XBeeDevice method), 250

send_bluetooth_data()
(digi.xbee.devices.ZigBeeDevice method), 335

send_data()
(digi.xbee.devices.CellularDevice method), 373

send_data()
(digi.xbee.devices.DigiMeshDevice method), 296

send_data()
(digi.xbee.devices.DigiPointDevice method), 315

send_data()
(digi.xbee.devices.IPDevice method), 347

send_data()
(digi.xbee.devices.LPWANDevice method), 392

send_data()
(digi.xbee.devices.NBIoTDevice method), 409

send_data()
(digi.xbee.devices.Raw802Device method), 277

send_data()
(digi.xbee.devices.WiFiDevice method), 432

send_data()
(digi.xbee.devices.XBeeDevice method), 249

send_data()
(digi.xbee.devices.ZigBeeDevice method), 335

send_data_16() (digi.xbee.devices.Raw802Device method), 266

send_data_64() (digi.xbee.devices.DigiMeshDevice method), 284

send_data_64() (digi.xbee.devices.Raw802Device method), 266

send_data_64_16() (digi.xbee.devices.DigiPointDevice method), 303

send_data_64_16() (digi.xbee.devices.ZigBeeDevice method), 323

send_data_async() (digi.xbee.devices.CellularDevice method), 373

send_data_async() (digi.xbee.devices.DigiMeshDevice method), 296

send_data_async() (digi.xbee.devices.DigiPointDevice method), 315

send_data_async() (digi.xbee.devices.IPDevice method), 347

send_data_async() (digi.xbee.devices.LPWANDevice method), 392

send_data_async() (digi.xbee.devices.NBIoTDevice method), 409

send_data_async() (digi.xbee.devices.Raw802Device method), 277

send_data_async() (digi.xbee.devices.WiFiDevice method), 432

send_data_async() (digi.xbee.devices.XBeeDevice method), 249

send_data_async() (digi.xbee.devices.ZigBeeDevice method), 336

send_data_async_16() (digi.xbee.devices.Raw802Device method), 266

send_data_async_64() (digi.xbee.devices.DigiMeshDevice method), 285

send_data_async_64() (digi.xbee.devices.Raw802Device method), 266

send_data_async_64_16() (digi.xbee.devices.DigiPointDevice method), 304

send_data_async_64_16() (digi.xbee.devices.ZigBeeDevice method), 323
send_data_broadcast()
(digi.xbee.devices.CellularDevice method), 374

send_data_broadcast()
(digi.xbee.devices.DigiMeshDevice method), 297

send_data_broadcast()
(digi.xbee.devices.DigiPointDevice method), 316

send_data_broadcast()
(digi.xbee.devices.IPDevice method), 347

send_data_broadcast()
(digi.xbee.devices.LPWANDevice method), 392

send_data_broadcast()
(digi.xbee.devices.NBIoTDevice method), 409

send_data_broadcast()
(digi.xbee.devices.Raw802Device method), 278

send_data_broadcast()
(digi.xbee.devices.WiFiDevice method), 433

send_expl_data()
(digi.xbee.devices.DigiMeshDevice method), 285

send_expl_data()
(digi.xbee.devices.DigiPointDevice method), 304

send_expl_data_async()
(digi.xbee.devices.DigiMeshDevice method), 285

send_expl_data_async()
(digi.xbee.devices.DigiPointDevice method), 304

send_expl_data_async()
(digi.xbee.devices.ZigBeeDevice method), 324

send_expl_data_async()
(digi.xbee.devices.ZigBeeDevice method), 324

send_ip_data()
(digi.xbee.devices.CellularDevice method), 374

send_ip_data()
(digi.xbee.devices.IPDevice method), 344

send_ip_data()
(digi.xbee.devices.LPWANDevice method), 380

send_ip_data()
(digi.xbee.devices.NBIoTDevice method), 409

send_ip_data()
(digi.xbee.devices.WiFiDevice method), 433

send_ip_data_async()
(digi.xbee.devices.CellularDevice method), 374

send_ip_data_async()
(digi.xbee.devices.IPDevice method), 345

send_ip_data_async()
(digi.xbee.devices.LPWANDevice method), 380

send_ip_data_async()
(digi.xbee.devices.NBIoTDevice method), 409

send_ip_data_async()
(digi.xbee.devices.WiFiDevice method), 433

send_ip_data_broadcast()
(digi.xbee.devices.CellularDevice method), 375

send_ip_data_broadcast()
(digi.xbee.devices.IPDevice method), 345

send_ip_data_broadcast()
(digi.xbee.devices.LPWANDevice method), 392

send_ip_data_broadcast()
(digi.xbee.devices.NBIoTDevice method), 409

send_ip_data_broadcast()
(digi.xbee.devices.WiFiDevice method), 434

send_micropython_data()
(digi.xbee.devices.CellularDevice method), 375

send_micropython_data()
(digi.xbee.devices.DigiMeshDevice method), 297

send_micropython_data()
(digi.xbee.devices.DigiPointDevice method), 316

send_micropython_data()
(digi.xbee.devices.IPDevice method), 356

send_micropython_data()
(digi.xbee.devices.LPWANDevice method), 392

send_micropython_data()
(digi.xbee.devices.NBIoTDevice method), 410

send_micropython_data()
(digi.xbee.devices.WiFiDevice method), 278
send_micropython_data()
(digi.xbee.devices.WiFiDevice method), 434
send_micropython_data()
(digi.xbee.devices.XBeeDevice method), 251
send_micropython_data()
(digi.xbee.devices.ZigBeeDevice method), 337
send_multicast_data()
(digi.xbee.devices.ZigBeeDevice method), 324
send_multicast_data_async()
(digi.xbee.devices.ZigBeeDevice method), 325
send_packet()
(digi.xbee.devices.CellularDevice method), 375
send_packet()
(digi.xbee.devices.DigiMeshDevice method), 298
send_packet()
(digi.xbee.devices.DigiPointDevice method), 317
send_packet()
(digi.xbee.devices.IPDevice method), 356
send_packet()
(digi.xbee.devices.LPWANDevice method), 393
send_packet()
(digi.xbee.devices.NBIoTDevice method), 410
send_packet()
(digi.xbee.devices.Raw802Device method), 279
send_packet()
(digi.xbee.devices.WiFiDevice method), 434
send_packet()
(digi.xbee.devices.XBeeDevice method), 254
send_packet_sync_and_get_response()
(digi.xbee.devices.XBeeDevice method), 337
send_sm()
(digi.xbee.devices.CellularDevice method), 362
send_sm()
(digi.xbee.devices.LPWANDevice method), 381
send_sm()
(digi.xbee.devices.NBIoTDevice method), 410
send_sm_async()
(digi.xbee.devices.CellularDevice method), 363
send_sm_async()
(digi.xbee.devices.LPWANDevice method), 381
send_sm_async()
(digi.xbee.devices.NBIoTDevice method), 410
send_user_data_relay()
(digi.xbee.devices.CellularDevice method), 376
send_user_data_relay()
(digi.xbee.devices.DigiMeshDevice method), 298
send_user_data_relay()
(digi.xbee.devices.DigiPointDevice method), 317
send_user_data_relay()
(digi.xbee.devices.IPDevice method), 356
send_user_data_relay()
(digi.xbee.devices.LPWANDevice method), 393
send_user_data_relay()
(digi.xbee.devices.NBIoTDevice method), 411
send_user_data_relay()
(digi.xbee.devices.NBIoTDevice method), 411
send_user_data_relay()
(digi.xbee.devices.Raw802Device method), 279
SendDataRequestOptions (class in digi.xbee.models.options), 110
SendDataRequestPacket (class in digi.xbee.packets.devicecloud), 184
SendDataResponsePacket (class in digi.xbee.packets.devicecloud), 187
SENT_FROM_END_DEVICE (digi.xbee.models.options.ReceiveOptions attribute), 108
serial_port (digi.xbee.devices.CellularDevice attribute), 376
serial_port (digi.xbee.devices.DigiMeshDevice at-
set_dest_address() (digi.xbee.devices.DigiPointDevice method), 318
set_dest_address() (digi.xbee.devices.IPDevice method), 346
set_dest_address() (digi.xbee.devices.LPWANDevice method), 394
set_dest_address() (digi.xbee.devices.NBIoTDevice method), 412
set_dest_address() (digi.xbee.devices.Raw802Device method), 280
set_dest_address() (digi.xbee.devices.RemoteDigiMeshDevice method), 472
set_dest_address() (digi.xbee.devices.RemoteDigiPointDevice method), 484
set_dest_address() (digi.xbee.devices.RemoteRaw802Device method), 460
set_dest_address() (digi.xbee.devices.RemoteXBeeDevice method), 448
set_dest_address() (digi.xbee.devices.RemoteZigBeeDevice method), 496
set_dest_address() (digi.xbee.devices.WiFiDevice method), 436
set_dest_address() (digi.xbee.devices.XBeeDevice method), 261
set_dest_address() (digi.xbee.devices.ZigBeeDevice method), 339
set_dest_ip_addr() (digi.xbee.devices.CellularDevice method), 377
set_dest_ip_addr() (digi.xbee.devices.IPDevice method), 343
set_dest_ip_addr() (digi.xbee.devices.LPWANDevice method), 394
set_dest_ip_addr() (digi.xbee.devices.NBIoTDevice method), 412
set_dest_ip_addr() (digi.xbee.devices.WiFiDevice method), 436
set_dio_change_detection() (digi.xbee.devices.AbstractXBeeDevice method), 245
set_dio_change_detection() (digi.xbee.devices.CellularDevice method), 363
set_dio_change_detection() (digi.xbee.devices.DigiMeshDevice method), 299
set_dio_change_detection() (digi.xbee.devices.DigiPointDevice method), 318
set_dio_change_detection() (digi.xbee.devices.IPDevice method), 358
set_dio_change_detection() (digi.xbee.devices.LPWANDevice method), 394
set_dio_change_detection() (digi.xbee.devices.NBIoTDevice method), 412
set_dio_change_detection() (digi.xbee.devices.RemoteDigiMeshDevice method), 472
set_dio_change_detection() (digi.xbee.devices.RemoteDigiPointDevice method), 484
set_dio_change_detection() (digi.xbee.devices.RemoteRaw802Device method), 460
set_dio_change_detection() (digi.xbee.devices.RemoteXBeeDevice method), 448
set_dio_change_detection() (digi.xbee.devices.RemoteZigBeeDevice method), 496
set_dio_value() (digi.xbee.devices.AbstractXBeeDevice method), 339
set_dio_value() (digi.xbee.devices.CellularDevice method), 244
set_dio_value() (digi.xbee.devices.CellularDevice method), 377
set_dio_value() (digi.xbee.devices.DigiMeshDevice method), 300
set_dio_value() (digi.xbee.devices.DigiPointDevice method), 319
set_dio_value() (digi.xbee.devices.IPDevice method), 358
set_dio_value() (digi.xbee.devices.LPWANDevice method), 394
set_dio_value() (digi.xbee.devices.NBIoTDevice method), 412
set_dio_value() (digi.xbee.devices.Raw802Device method), 281
set_dio_value() (digi.xbee.devices.RemoteDigiMeshDevice method), 472
set_dio_value() (digi.xbee.devices.RemoteDigiPointDevice method), 484
set_dio_value() (digi.xbee.devices.RemoteRaw802Device method), 460
set_dio_value() (digi.xbee.devices.RemoteXBeeDevice method), 449
set_dio_value() (digi.xbee.devices.RemoteZigBeeDevice method), 496
set_dio_value() (digi.xbee.devices.ZigBeeDevice method), 339
set_discovery_options() (digi.xbee.devices.DigiMeshNetwork method), 520
set_discovery_options() (digi.xbee.devices.DigiPointNetwork method), 525
set_discovery_options() (digi.xbee.devices.Raw802Network method), 514
set_discovery_options() (digi.xbee.devices.XBeeNetwork method), 503
set_discovery_options() (digi.xbee.devices.ZigBeeNetwork method), 509
set_discovery_timeout() (digi.xbee.devices.DigiMeshNetwork method), 520
set_discovery_timeout() (digi.xbee.devices.DigiPointNetwork method), 525
set_discovery_timeout() (digi.xbee.devices.Raw802Network method), 515
set_discovery_timeout() (digi.xbee.devices.XBeeNetwork method), 503
set_discovery_timeout() (digi.xbee.devices.ZigBeeNetwork method), 510
set_dns_address() (digi.xbee.devices.ZigBeeDevice method), 340
set_dns_address() (digi.xbee.devices.XBeeDevice method), 342
set_dns_address() (digi.xbee.devices.ZigBeeDevice method), 342
set_gateway_address() (digi.xbee.devices.ZigBeeDevice method), 421
set_gateway_address() (digi.xbee.devices.XBeeDevice method), 421
set_io_configuration() (digi.xbee.devices.AbstractXBeeDevice method), 241
set_io_configuration() (digi.xbee.devices.CellularDevice method), 378
set_io_configuration() (digi.xbee.devices.DigiMeshDevice method), 413
set_io_configuration() (digi.xbee.devices.IPDevice method), 358
set_io_configuration() (digi.xbee.devices.LPWANDevice method), 395
set_io_configuration() (digi.xbee.devices.NBIoTDevice method), 449
set_io_configuration() (digi.xbee.devices.ZigBeeDevice method), 340
set_io_sampling_rate() (digi.xbee.devices.ZigBeeDevice method), 342
set_io_sampling_rate() (digi.xbee.devices.XBeeDevice method), 342
set_io_sampling_rate() (digi.xbee.devices.ZigBeeDevice method), 342
set_io_sampling_rate() (digi.xbee.devices.XBeeDevice method), 349
set_io_sampling_rate() (digi.xbee.devices.XBeeDevice method), 349
set_io_sampling_rate() (digi.xbee.devices.ZigBeeDevice method), 349
set_io_sampling_rate()
(digi.xbee.devices.DigiPointDevice method), 320
set_io_sampling_rate()
(digi.xbee.devices.IPDevice method), 359
set_io_sampling_rate()
(digi.xbee.devices.LPWANDevice method), 395
set_io_sampling_rate()
(digi.xbee.devices.NBIoTDevice method), 413
set_io_sampling_rate()
(digi.xbee.devices.Raw802Device method), 282
set_io_sampling_rate()
(digi.xbee.devices.RemoteDigiMeshDevice method), 473
set_io_sampling_rate()
(digi.xbee.devices.RemoteDigiPointDevice method), 485
set_io_sampling_rate()
(digi.xbee.devices.RemoteRaw802Device method), 462
set_io_sampling_rate()
(digi.xbee.devices.RemoteXBeeDevice method), 450
set_io_sampling_rate()
(digi.xbee.devices.RemoteZigBeeDevice method), 498
set_io_sampling_rate()
(digi.xbee.devices.WiFiDevice method), 437
set_io_sampling_rate()
(digi.xbee.devices.XBeeDevice method), 263
set_io_sampling_rate()
(digi.xbee.devices.ZigBeeDevice method), 340
set_ip_address()
(digi.xbee.devices.WiFiDevice method), 420
set_ip_addressing_mode()
(digi.xbee.devices.WiFiDevice method), 420
set_local_xbee_device()
(digi.xbee.devices.RemoteDigiMeshDevice method), 473
set_local_xbee_device()
(digi.xbee.devices.RemoteDigiPointDevice method), 485
set_local_xbee_device()
(digi.xbee.devices.RemoteRaw802Device method), 462
set_local_xbee_device()
(digi.xbee.devices.RemoteXBeeDevice method), 441
set_local_xbee_device()
(digi.xbee.devices.RemoteZigBeeDevice method), 498
set_mask_address()
(digi.xbee.devices.WiFiDevice method), 420
set_node_id()
(digi.xbee.devices.AbstractXBeeDevice method), 238
set_node_id()
(digi.xbee.devices.CellularDevice method), 364
set_node_id()
(digi.xbee.devices.DigiMeshDevice method), 301
set_node_id()
(digi.xbee.devices.DigiPointDevice method), 320
set_node_id()
(digi.xbee.devices.IPDevice method), 359
set_node_id()
(digi.xbee.devices.LPWANDevice method), 395
set_node_id()
(digi.xbee.devices.NBIoTDevice method), 413
set_node_id()
(digi.xbee.devices.Raw802Device method), 282
set_node_id()
(digi.xbee.devices.RemoteDigiMeshDevice method), 474
set_node_id()
(digi.xbee.devices.RemoteDigiPointDevice method), 486
set_node_id()
(digi.xbee.devices.RemoteRaw802Device method), 462
set_node_id()
(digi.xbee.devices.RemoteXBeeDevice method), 450
set_node_id()
(digi.xbee.devices.RemoteZigBeeDevice method), 498
set_pan_id()
(digi.xbee.devices.AbstractXBeeDevice method), 240
set_pan_id()
(digi.xbee.devices.CellularDevice method), 378
set_pan_id()
(digi.xbee.devices.DigiMeshDevice method), 301
set_pan_id()
(digi.xbee.devices.DigiPointDevice method), 320
set_pan_id()
(digi.xbee.devices.IPDevice method), 346
set_pan_id()
(digi.xbee.devices.LPWANDevice method), 395
set_pan_id()
(digi.xbee.devices.NBIoTDevice method), 413
set_pan_id()
(digi.xbee.devices.Raw802Device method), 282
set_pan_id()
(digi.xbee.devices.RemoteDigiMeshDevice
set_pan_id() (digi.xbee.devices.RemoteDigiPointDevice method), 486
set_pan_id() (digi.xbee.devices.RemoteRaw802Device method), 462
set_pan_id() (digi.xbee.devices.RemoteXBeeDevice method), 450
set_pan_id() (digi.xbee.devices.RemoteZigBeeDevice method), 498
set_pan_id() (digi.xbee.devices.WiFiDevice method), 438
set_pan_id() (digi.xbee.devices.XBeeDevice method), 263
set_pan_id() (digi.xbee.devices.ZigBeeDevice method), 341
set_parameter() (digi.xbee.devices.AbstractXBeeDevice method), 236
set_parameter() (digi.xbee.devices.CellularDevice method), 378
set_parameter() (digi.xbee.devices.DigiMeshDevice method), 301
set_parameter() (digi.xbee.devices.DigiPointDevice method), 320
set_parameter() (digi.xbee.devices.IPDevice method), 359
set_parameter() (digi.xbee.devices.LPWANDevice method), 396
set_parameter() (digi.xbee.devices.NBIoTDevice method), 413
set_parameter() (digi.xbee.devices.Raw802Device method), 282
set_power_level() (digi.xbee.devices.DigiMeshDevice method), 301
set_power_level() (digi.xbee.devices.DigiPointDevice method), 320
set_power_level() (digi.xbee.devices.IPDevice method), 359
set_power_level() (digi.xbee.devices.LPWANDevice method), 396
set_power_level() (digi.xbee.devices.NBIoTDevice method), 413
set_power_level() (digi.xbee.devices.Raw802Device method), 282
set_power_level() (digi.xbee.devices.RemoteDigiMeshDevice method), 474
set_power_level() (digi.xbee.devices.RemoteDigiPointDevice method), 486
set_power_level() (digi.xbee.devices.RemoteRaw802Device method), 462
set_power_level() (digi.xbee.devices.RemoteXBeeDevice method), 450
set_power_level() (digi.xbee.devices.RemoteZigBeeDevice method), 498
set_power_level() (digi.xbee.devices.WiFiDevice method), 438
set_power_level() (digi.xbee.devices.XBeeDevice method), 263
set_power_level() (digi.xbee.devices.ZigBeeDevice method), 341
set_pwm_duty_cycle() (digi.xbee.devices.AbstractXBeeDevice method), 243
set_pwm_duty_cycle() (digi.xbee.devices.CellularDevice method), 378
set_pwm_duty_cycle() (digi.xbee.devices.DigiMeshDevice method), 301
set_pwm_duty_cycle() (digi.xbee.devices.DigiPointDevice method), 320
set_pwm_duty_cycle() (digi.xbee.devices.IPDevice method), 359
set_pwm_duty_cycle() (digi.xbee.devices.LPWANDevice method), 396
set_pwm_duty_cycle()
(digi.xbee.devices.NBIoTDevice method), 396

set_pwm_duty_cycle()
(digi.xbee.devices.Raw802Device method), 413

set_pwm_duty_cycle()
(digi.xbee.devices.RemoteDigiMeshDevice method), 474

set_pwm_duty_cycle()
(digi.xbee.devices.RemoteDigiPointDevice method), 487

set_pwm_duty_cycle()
(digi.xbee.devices.RemoteRaw802Device method), 463

set_pwm_duty_cycle()
(digi.xbee.devices.RemoteXBeeDevice method), 451

set_pwm_duty_cycle()
(digi.xbee.devices.RemoteZigBeeDevice method), 499

set_pwm_duty_cycle()
(digi.xbee.devices.WiFiDevice method), 439

set_read_timeout()
(digi.xbee.serial.XBeeSerialPort method), 552

set_sync_ops_timeout()
(digi.xbee.devices.AbstractXBeeDevice method), 240

set_sync_ops_timeout()
(digi.xbee.devices.CellularDevice method), 379

set_sync_ops_timeout()
(digi.xbee.devices.DigiMeshDevice method), 302

set_sync_ops_timeout()
(digi.xbee.devices.DigiPointDevice method), 321

set_sync_ops_timeout()
(digi.xbee.devices.IPDevice method), 360

set_sync_ops_timeout()
(digi.xbee.devices.LPWANDevice method), 396

set_sync_ops_timeout()
(digi.xbee.devices.NBIoTDevice method), 414

set_sync_ops_timeout()
(digi.xbee.devices.Raw802Device method), 283

set_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiMeshDevice method), 475

set_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiPointDevice method), 487

set_sync_ops_timeout()
(digi.xbee.devices.RemoteRaw802Device method), 463

set_sync_ops_timeout()
(digi.xbee.devices.RemoteXBeeDevice method), 451

set_sync_ops_timeout()
(digi.xbee.devices.RemoteZigBeeDevice method), 499

set_sync_ops_timeout()
(digi.xbee.devices.WiFiDevice method), 439

signal_quality(digi.xbee.models.accesspoint.AccessPoint attribute), 95

set_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiMeshDevice method), 474

set_sync_ops_timeout()
(digi.xbee.devices.RemoteDigiPointDevice method), 486

set_sync_ops_timeout()
(digi.xbee.devices.RemoteRaw802Device method), 462

set_sync_ops_timeout()
(digi.xbee.devices.RemoteXBeeDevice method), 450

set_sync_ops_timeout()
(digi.xbee.devices.RemoteZigBeeDevice method), 498

set_sync_ops_timeout()
(digi.xbee.devices.WiFiDevice method), 438

sort()
(digi.xbee.reader.BluetoothDataReceived method), 543

sort()
(digi.xbee.reader.DataReceived method), 535

sort()
(digi.xbee.reader.DeviceDiscovered method), 538

sort()
(digi.xbee.reader.DiscoveryProcessFinished method), 539

sort()
(digi.xbee.reader.ExplicitDataReceived method), 540

sort()
(digi.xbee.reader.IODataReceived method), 541

sort()
(digi.xbee.reader.IOSampleReceived method), 542

sort()
(digi.xbee.reader.ModemStatusReceived method), 536

sort()
(digi.xbee.reader.PacketReceived method), 534

sort()
(digi.xbee.reader.RelayDataReceived method), 544

sort()
(digi.xbee.reader.XBeeEvent method), 533

source_address(digi.xbee.packets.network.RXIPv4Packet attribute), 191

source_address(digi.xbee.packets.wifi.IODataSampleRxIndicatorWifi attribute), 223

Index 605
source_address (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket attribute), 231
source_endpoint (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 168
source_endpoint (digi.xbee.packets.wifi.RemoteATCommandResponseWifiPacket attribute), 231
source_port (digi.xbee.models.message.IPMessage attribute), 106
source_port (digi.xbee.packets.network.RXIPv4Packet attribute), 191
source_port (digi.xbee.packets.network.TXIPv4Packet attribute), 196
SPECIAL_FUNCTIONALITY (digi.xbee.io.IOMode attribute), 532
SpecialByte (class in digi.xbee.models.atcomm), 96
src_interface (digi.xbee.packets.relay.UserDataRelayOutputPacket attribute), 221
ssid (digi.xbee.models.accesspoint.AccessPoint attribute), 95
start() (digi.xbee.reader.PacketListener method), 548
start_discovery_process() (digi.xbee.devices.DigiMeshNetwork method), 520
start_discovery_process() (digi.xbee.devices.DigiPointNetwork method), 526
start_discovery_process() (digi.xbee.devices.Raw802Network method), 515
start_discovery_process() (digi.xbee.devices.XBeeNetwork method), 500
start_discovery_process() (digi.xbee.devices.ZigBeeNetwork method), 510
start_listening() (digi.xbee.devices.CellularDevice method), 379
start_listening() (digi.xbee.devices.IPDevice method), 344
start_listening() (digi.xbee.devices.LPWANDevice method), 396
start_listening() (digi.xbee.devices.NBIoTDevice method), 414
start_listening() (digi.xbee.devices.WiFiDevice method), 439
target (digi.xbee.packets.devicecloud.DeviceRequestPacket attribute), 175
task_done() (digi.xbee.reader.XBeeQueue method), 551
TIMEOUT_READ_PACKET (digi.xbee.devices.XBeeDevice attribute), 248
TimeoutException, 528
timestamp (digi.xbee.models.message.ExplicitXBeeMessage attribute), 105
timestamp (digi.xbee.models.message.XBeeMessage attribute), 104
to_dict() (digi.xbee.models.message.ExplicitXBeeMessage method), 104
to_dict() (digi.xbee.models.message.IPMessage method), 105
to_dict() (digi.xbee.models.message.SMSMessage method), 106
Transport

TX16Packet (class in digi.xbee.packets.raw), 199
TX64Packet (class in digi.xbee.packets.raw), 196
TXIPv4Packet (class in digi.xbee.packets.network), 193
TXSMSPacket (class in digi.xbee.packets.cellular), 133
TXStatusPacket (class in digi.xbee.packets.raw), 202

unescape_data() (digi.xbee.packets.devicecloud.DeviceRequestPacket static method), 175
unescape_data() (digi.xbee.packets.devicecloud.DeviceResponsePacket static method), 182
unescape_data() (digi.xbee.packets.devicecloud.FrameErrorPacket static method), 187
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 189
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 193
unescape_data() (digi.xbee.packets.devicecloud.DeviceResponseStatusPacket static method), 196
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 207
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 207
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 215
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 219
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 225
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 228
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 231
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 238
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 247
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 252
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 269
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 279
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 298
unescape_data() (digi.xbee.packets.devicecloud.SendDataResponsePacket static method), 302
unescape_data() (digi.xbee.packets.devicecloud.SendDataRequestPacket static method), 311
update_bluetooth_password()
(digi.xbee.devices.IPDevice method), 360
update_bluetooth_password()
(digi.xbee.devices.LPWANDevice method), 397
update_bluetooth_password()
(digi.xbee.devices.NBIoTDevice method), 414
update_bluetooth_password()
(digi.xbee.devices.Raw802Device method), 283
update_bluetooth_password()
(digi.xbee.devices.RemoteDigiMeshDevice method), 475
update_bluetooth_password()
(digi.xbee.devices.RemoteDigiPointDevice method), 487
update_bluetooth_password()
(digi.xbee.devices.RemoteRaw802Device method), 463
update_bluetooth_password()
(digi.xbee.devices.RemoteXBeeDevice method), 451
update_bluetooth_password()
(digi.xbee.devices.RemoteZigBeeDevice method), 499
update_bluetooth_password()
(digi.xbee.devices.WiFiDevice method), 439
update_bluetooth_password()
(digi.xbee.devices.XBeeDevice method), 264
update_bluetooth_password()
(digi.xbee.devices.ZigBeeDevice method), 342
update_device_data_from()
(digi.xbee.devices.AbstractXBeeDevice method), 236
update_device_data_from()
(digi.xbee.devices.CellularDevice method), 379
update_device_data_from()
(digi.xbee.devices.DigiMeshDevice method), 302
update_device_data_from()
(digi.xbee.devices.DigiPointDevice method), 321
update_device_data_from()
(digi.xbee.devices.IPDevice method), 360
update_device_data_from()
(digi.xbee.devices.LPWANDevice method), 397
update_device_data_from()
(digi.xbee.devices.NBIoTDevice method), 415
update_device_data_from()
(digi.xbee.devices.Raw802Device method), 283
update_device_data_from()
(digi.xbee.devices.RemoteDigiMeshDevice method), 475
update_device_data_from()
(digi.xbee.devices.RemoteDigiPointDevice method), 487
update_device_data_from()
(digi.xbee.devices.RemoteRaw802Device method), 463
update_device_data_from()
(digi.xbee.devices.RemoteXBeeDevice method), 451
update_device_data_from()
(digi.xbee.devices.RemoteZigBeeDevice method), 499
update_device_data_from()
(digi.xbee.devices.WiFiDevice method), 439
update_device_data_from()
(digi.xbee.devices.XBeeDevice method), 264
update_device_data_from()
(digi.xbee.devices.ZigBeeDevice method), 342
USE_BROADCAST_PAN_ID
(digi.xbee.models.options.TransmitOptions attribute), 108
USE_EXTENDED_TIMEOUT
(digi.xbee.models.options.TransmitOptions attribute), 109
WiFiAssociationIndicationStatus
(class in digi.xbee.models.status), 120
WiFiDevice
(class in digi.xbee.devices), 415
WiFiEncryptionType
(class in digi.xbee.models.accesspoint), 95
with_traceback()
(digi.xbee.exception.ATCommandException method), 526
with_traceback()
(digi.xbee.exception.CommunicationException method), 526
with_traceback()
(digi.xbee.exception.ConnectionException method), 527
with_traceback()
(digi.xbee.exception.InvalidConfigurationException method), 527

Index
XBee Python Library Documentation, Release 1.2.0

with_traceback() (digi.xbee.exception.OperatingModeException method), 527
with_traceback() (digi.xbee.exception.TimeoutException)_dest_addr (digi.xbee.packets.common.TransmitStatusPacket attribute), 159
with_traceback() (digi.xbee.exception.DataLinkLayerException) 527
write_changes() (digi.xbee.devices.AbstractXBeeDevice x16bit_source_addr, 200
write_changes() (digi.xbee.devices.CellularDevice x16bit_source_addr, 152
write_changes() (digi.xbee.devices.DigiMeshDevice x16bit_source_addr, 216
write_changes() (digi.xbee.devices.DigiPointDevice x16bit_source_addr, 146
write_changes() (digi.xbee.devices.IPDevice x64bit_dest_addr (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 168
write_changes() (digi.xbee.devices.LPWANDevice x64bit_dest_addr (digi.xbee.packets.common.RemoteATCommandPacket attribute), 149
write_changes() (digi.xbee.devices.NBIoTDevice x64bit_dest_addr (digi.xbee.packets.common.TransmitPacket attribute), 156
write_changes() (digi.xbee.devices.Raw802Device x64bit_dest_addr (digi.xbee.packets.common.ReceivePacket attribute), 197
write_changes() (digi.xbee.devices.RemoteDigiMeshDevice x64bit_source_addr, 173
write_changes() (digi.xbee.devices.RemoteDigiPointDevice x64bit_source_addr, 165
write_changes() (digi.xbee.devices.RemoteRaw802Device x64bit_source_addr, 146
write_changes() (digi.xbee.devices.RemoteXBeeDevice x64bit_source_addr, 152
write_changes() (digi.xbee.devices.ZigBeeDevice x64bit_source_addr, 152
write_changes() (digi.xbee.devices.WiFiDevice x64bit_source_addr, 206
write_changes() (digi.xbee.devices.XBeeDevice x64bit_source_addr, 206
x16bit_dest_addr (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 168
x16bit_dest_addr (digi.xbee.packets.common.RemoteATCommandPacket attribute), 149
x16bit_dest_addr (digi.xbee.packets.common.TransmitStatusPacket attribute), 156
x64bit_dest_addr (digi.xbee.packets.common.ExplicitAddressingPacket attribute), 146
x64bit_dest_addr (digi.xbee.packets.common.RemoteATCommandPacket attribute), 152
x64bit_dest_addr (digi.xbee.packets.common.TransmitPacket attribute), 156
x64bit_dest_addr (digi.xbee.packets.common.ReceivePacket attribute), 197

XBee16BitAddress (class in digi.xbee.models.address), 101
XBee64BitAddress (class in digi.xbee.models.address), 102
XBeeAPIPacket (class in digi.xbee.packets.base), 123
XBeeDevice (class in digi.xbee.devices), 247
XBeeDeviceException, 527
XBeeEvent (class in digi.xbee.reader), 533
XBeeException, 526
XBeeIMEIAddress (class in digi.xbee.models.address), 103
XBeeLocalInterface (class in digi.xbee.models.options), 111
XBeeMessage (class in digi.xbee.models.message), 103
XBeeNetwork (class in digi.xbee.devices), 500
XBeePacket (class in digi.xbee.packets.base), 122
XBeeProtocol (class in digi.xbee.models.protocol), 112
XBeeQueue (class in digi.xbee.reader), 548
XBeeSerialPort (class in digi.xbee.serial), 551

Z
ZigBeeDevice (class in digi.xbee.devices), 322
ZigBeeNetwork (class in digi.xbee.devices), 505